Process의 특성을 반영한 Experiential Scheduler

오만석
홍익대학교 정보컴퓨터 공학부 컴퓨터공학
msj624@naver.com

Development of the Experiential Scheduler Based on the Features of Process

Manseok Oh
Information & Computer Engineering University of Hongik
msj624@naver.com

요 약

현재 리눅스에서 사용되고 있는 스케줄링 방식은 Weight(가중치), Quantum(기본 설정 CPU점유시간) 등을 이용하여 스케줄링 한다. 이러한 스케줄링 방식으로는 프로세스 각각의 특징을 반영하기 힘들다.
이러한 점을 개선하고자 하나의 프로세스가 실행될 때, 실행 프로세스의 특징을 경험적 데이터로 저장함으로서, 프로세스가 다시 실행될 시에, 경험적 데이터를 적용하여 프로세스의 특징을 스케줄링에 반영토록 한다.
경험적인 데이터들로는 프로세스의 실행시간, 프로세스의 종류, 실행 빈도 등이 있는데, 이들을 스케줄링에 적용하여, 프로세스 각각의 특징을 반영하여 각각의 프로세스에 최적화된 스케줄러를 구현한다. 개발 대상 OS는 Open Source이며, 다양한 분야에 적용되고 있는 리눅스를 선정하였고, 기존의 리눅스 스케줄링에 과거의 경험적인 데이터를 반영하여, 좋은 효율의 스케줄러를 구현하는데 목적을 두었다.

1. 서 론

오늘날 OS는 많은 관련 알고리즘들이 연구되고, 개발되었다. 그 중에서 스케줄링은 OS의 핵심이 되는 부분이다. 스케줄링 관련 알고리즘 중에는 실제 구현되어 사용되고 있는 알고리즘들이 있는가 하면, 이들 중 연구는 되었으나, 구현되지 못한 알고리즘들도 있다. 이러한 알고리즘 중에서 SJF(Shortest Job First)는 앞으로 실행될 프로세스의 실행시간에 대한 예측이 불가능하기 때문에 구현되지 않은 스케줄링 알고리즘이다.[1] 이 알고리즘에서 아이디어를 얻었고 현재의 리눅스 스케줄러에 프로세스들의 경험적 데이터를 반영하여 각 프로세스 마다 최적의 time_slice와 우선순위 값을 준 결과 현재의 리눅스 스케줄러보다 좋은 효율을 나타 낼 수 있었다.[2][3]

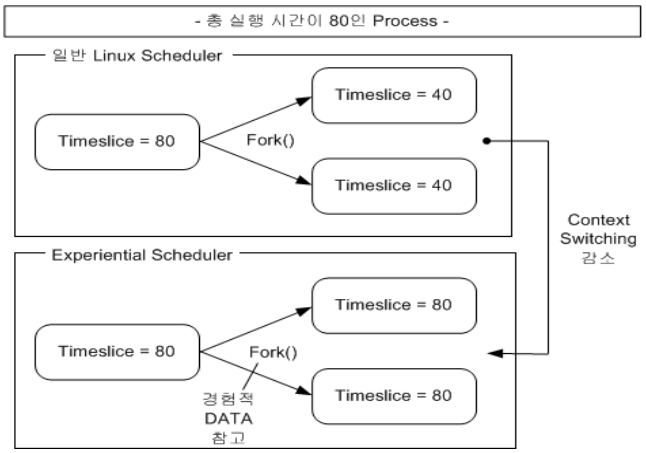

그림 1 . Experiential Scheduler Concept Example

2. 전체 시스템 구성 및 특징

전체 시스템은 크게 커널과 모듈로 부분으로 나눌 수 있다. 커널 내부적으로는 프로세스 생성과 소멸 시 프로세스의 경험적 데이터 송·수신부와 실제 스케줄에 적용하는 스케줄러가 있고, 모듈은 경험적 데이터를 저장하는 테이블과 테이블의 초기화 및 관리를 하는 부분으로 모듈을 나눌 수 있다. 앞으로 이 모듈은 Experiential Module이라 칭한다.

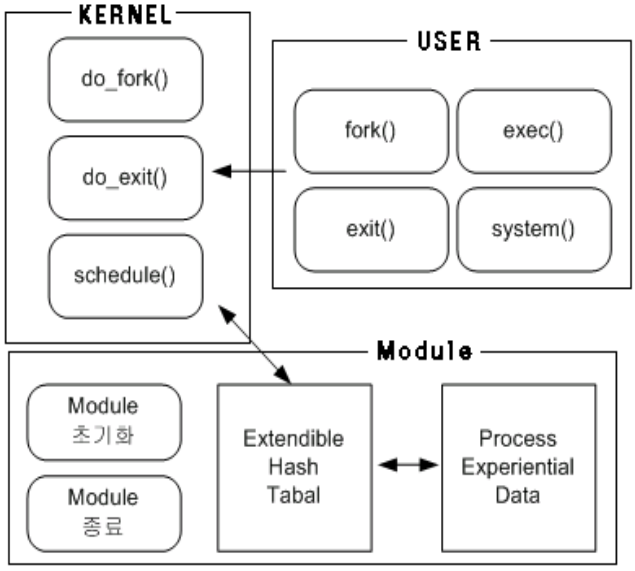

그림 2 . 전제 시스템 구성

2.1 각 구현 부분 특징

2.1 Kernel 구현

기존 리눅스 커널-2.6.18 의 스케줄러를 수정하여 경험적 데이터를 스케줄러에 반영한다. 부분별로 나누면 프로세스 생성 부분과 프로세스 종료부분, schedule()로 나눌 수 있다.

2.1.1 경험적 데이터 구조체

```
typedef struct
{
  // process의 이름 변수
     char Experiential_process_name[TASK LEN];

  // process의 timeslice 변수
     unsigned int Experiential_time_slice;

  // process의 timestamp 변수
     unsigned int Experiential_timestamp;

  // process의 type 변수
     unsigned char Experiential_process_type;

  // process의 우선순위를 위한 변수
     unsigned char Experiential_process_frequency;

} __attribute__((packed)) Experiential_DATA;
```

위의 구조체는 프로세스가 경험적 데이터를 사용하는데 이용 된다. Experiential_time_slice, Experiential_time_stamp등은 기본 스케줄러에서 사용되는 데이터와 같은 용도이다. Experiential_process_type와 Experiential_process_frequency는 기존에 없는 데이터인데 프로세스 타입과 실행빈도를 의미한다. process_name을 저장하는 변수에서 TASK LEN은 기존 스케줄러에서 사용하는 16과 동일하게 define하여 호환성을 유지하였다.

2.1.2 프로세스 생성 부분

프로세스 생성 시 extendible hash table의 경험적 데이터를 검색한다. 검색 시 process_name을 이용하게 되고, 만약 해당 프로세스에 대한 경험적 데이터가 존재한다면 프로세스와 관련된 경험적 데이터를 스케줄러에서 적용할 수 있게 값을 참조한다.

2.1.2.1 copy_process()

copy_process는 프로세스가 부모로부터 복사되어 하나의 프로세스가 만들어지는 과정의 함수이다. 위의 함수에서 task_struct에 할당되어있는 Experiential_DATA 구조체의 초기화를 담당한다.

2.1.2.2 EXPORT_SYMBOL

EXPORT_SYMBOL은 커널 Symbol 테이블에 변수 혹은 함수들을 등록하는 매크로이다. Symbol 테이블에 등록되어있는 변수나 함수는 extern으로 선언되어 커널 전역에서 자유롭게 접근이 가능하다. 이를 이용하여 최소한의 비용으로 쉽게 데이터 통신이 가능하게 했다.

```
/* using Experiential scheduler add symbol */
extern int lookup_item();
hash_table *p_hash_table = NULL;
EXPORT_SYMBOL(p_hash_table)
```

위의 소스는 exec.c에서 선언한 부분이다. lookup_item 함수는 extendible hash table에서 사용하는 함수로 자세한 설명은 이후에 하도록 하겠다. 선언 부분에 extern으로 선언하여 Symbol 테이블에 등록 되어있는 함수를 사용하는 구문이다. 이후 extendible hash table의 포인터를 선언하고, 아래 EXPORT_SYBMOL을 이용하여 커널 Symbol 테이블에 등록하였다.

2.1.2.3 Experiential_exec()

```
int Experiential_exec()
{
 ...
 // 현재 process의 name을 얻어 Experiential_DATA의
    process_name 변수에 값을 대입.

 // hash table에서 해당 process의 name을 이용하여
   hash table에 해당 process의 과거 정보가 있는지
   확인하여 해당 process의 time_slice값과 type값을
   바꾸어 준다.
 ...
}
```

위의 Experiential_exec 함수는 따로 구현한 함수로 역할은 extendible hash table에 있는 프로세스의 경험적 데이터를 해당 프로세스가 실행될 때 적용하는 것이다. Experiential_exec 함수는 flush_old_exec함수 안에서 호출하는데 flush_old_exec는 부모 프로세스로부터 같은 process_name을 할당받은 자식 프로세스가 자신만의 process_name을 얻게 되는 부분이다. 이때부터 자식 프로세스는 하나의 독립적인 프로세스로서 역할을 하게 된다.

2.1.3 프로세스 종료 부분

프로세스 소멸 시 extendible hash table에 경험적 데이터를 저장한다. 이 후 같은 프로세스 즉, 같은 process_name의 프로세스 생성 시 해당 프로세스는 경험적 데이터를 적용하게 된다.

2.1.3.1 do_exit()

```
fastcall NORET_TYPE void do_exit(long code) {

// 현재 process로 부터 경험적 데이터를 얻어 extendible
   hash_table에 저장.
if(add_item(process_data, p_hash_table)== -1)
    printk("[KERNEL] : ADD_ITEM ERROR₩n");
}
```

do_exit() 함수는 프로세스 종료시 호출되는 함수이다. 위의 함수에서 프로세스 실행 시 만들어진 경험적 데이터를 table에 추가한다. add_item함수는 차후 extendible hash table과 함께 설명하겠지만, extendible hash table에 경험적 데이터를 추가하는 함수이다. 위의 함수에서 인자로 사용되는 p_hash_table은 extendible hash table의 포인터인데 커널 심볼 테이블에 추가하여 extern 선언 후 커널 전역에서 사용할 수 있도록 하였다. 최초 설계시에는 queue를 이용하여 커널과 hash table간의 통신을 구현하였지만, 테스트 결과 오버헤드가 심하게 발생하여 위와 같은 방법으로 설계를 변경하여 구현하였다.

2.1.4 schedule()

task_struct에 들어있는 경험적 데이터를 스케줄에 적용하는 부분을 추가 수정하였다.

2.1.4.1 scheduler_tick()

```
void scheduler_tick(void) {
 ...
 /* process의 time_stamp 값을 +1만큼 증가 시켜준다.
    이 time_stamp의 총 합은 프로세스가 종료 시 해당
    프로세스의 time_slice값이 되어 extendible hash
    table에 저장 되어 이용된다.
 */
 Experiential_process_timestamp ++;
 ...
}
```

scheduler_tick는 실제 스케줄러 동작이 이루어지는 함수이다. 위의 함수의 특징은 하나의 tick에 함수가 종료되어야 한다. 위의 함수에서는 프로세스가 실행 되면서 경험적 데이터를 해당 선언 데이터에 추가한다. 이후 프로세스 종료 시 테이블에 추가한다.

2.1.4.1 effective_prio()

```
static int effective_prio(struct task_struct *process) {
   ...
   // 현재 process가 network나 멀티미디어 process인
      경우 해당 process의 우선순위 가중치 값을 100
      으로 설정
   ...
}
```

effective_prio 함수의 역할은 프로세스 각각의 우선순위에 대한 가중치를 적용하는 것인데 이 함수 안에서 해당 프로세스의 type 값을 확인하여, Experiential_NORMAL, 즉 일반 Process가 아닌 다른 특성을 가지는 Network Process나 Multimedia Process의 경우 우선순위에 대한 가중치를 RealTime Process와 일반 Process의 경계 값인 100으로 바꾸어 주어 일반 Process 보다 우선적인 실행을 보장해 준다. 그리고 일반 Process의 경우에는 실행 빈도 수에 따라 높은 우선순위를 주기 위해 해당 Process가 종료되는 시점에서 경험적 데이터를 extendible hash table에 저장할 때마다 우선순위에 대한 가중치를 증가 시켜 주었다. 모든 Process의 우선순위에 대한 가중치 값은 100을 초과 할 수 없게 하였다.

```
// Experiential process type
#define Experiential_NORMAL          1
#define Experiential_NETWORK         2
#define Experiential_MULTIMEDIA      4
```

프로세스 타입의 경우는 위와 같이 선언하였고 검색 시 정수 비교보다 빠른 비트 연산을 하기 위해 비트 단위로 선언하였다.

2.2 Experiential Module 구현

Experiential 모듈은 스케줄러에 필요한 테이블 할당 및 flag설정, register 등록 등의 초기화 과정을 수행하고 커널에서 모듈 해제 시 에는 메모리 해제 등의 역할을 한다.

2.2.1 모듈 초기화 함수 : Experiential_init()

Experiential 모듈이 커널에 적재될 시에 호출되는 함수로써 함수 내부적으로는 경험적 스케줄러에서 경험적 데이터 저장 역할을 하는 테이블에 대한 메모리 할당을 담당하고 경험적 스케줄러에 필요한 경험적 데이터의 초기화 역할을 한다. 구현 내용은 다음과 같다.

```
int __init Experiential_init(void) {

  /* initialize */
  // register 등록

  // extendible hash table 생성

  // 모듈 등록 확인을 위한 flag 설정.
}
```

구현 내용은 레지스터 등록 및 extendible hash table 생성, 모듈 등록여부 확인 변수 설정이다. 모듈 등록 확인의 경우에는 filp_open을 통해 해당 디바이스 파일을 open함으로서 확인할 수 있지만, 그럴 경우 파일 open

에 대한 비용이 큰 오버헤드로 발생 하여 성능저하를 유발하기 때문에 위와 같은 방법으로 구현 하였다.

2.2.1 모듈 종료 함수 : Experiential_exit()

위의 Experiential_exit는 모듈이 커널에서 해제 시에 호출되는 함수로 해제 이후에는 experiential scheduler를 사용하지 않기 때문에 불필요한 extendible hash table등의 메모리 해제 및 관리 역할을 한다. 구현 내용은 다음과 같다.

```
void __exit Experiential_exit(void) {
    // extendible hash table에 관련된 메모리를 해제한다.
    Experiential_purge_table(p_hash_table);

    // register에서 해제 한다.

    // 모듈 등록 여부에 대한 flag를 끈다.
}
```

2.3. Table

테이블은 extendible hash table중 메모리 관리를 효율적으로 할 수 있게 extendible hash table로 설계하고 구현하였다. process_name의 ASCII의 합을 100으로 나눈 값을 hashing에서의 key로 사용하였고 extendible hash의 특성에 맞게 최초 비교는 2개만을 사용하고, overflow 발생 시 하나씩 점차적으로 비트수를 늘리며 비교하였다. table은 세부적으로 크게 데이터 입력 부분, 데이터 출력 부분으로 나눌 수 있다.

- Experiential_hash_table()
  extendible hash table을 생성하여 준다.

- Experiential_purge_table()
  extendible hash table의 메모리 할당을 해제한다.

- 경험적 데이터 입력 부분 : Experiential_add_item()
  해당 프로세스가 종료 할 때의 상태를 받아서 경험적 데이터가 없는 경우 경험적 데이터를 설정해 주고 경험적 데이터가 있는 경우에는 timeslice, frequency, process_type 등의 Update를 담당한다.

- 경험적 데이터 출력 부분 : Experiential_lookup_item()
  해당 프로세스에 대한 경험적 데이터가 존재 하는지 확인하고 존재한다면 경험적 데이터를 보내준다.

위 함수들을 이용하여 테이블의 경험적 데이터를 이용한다. Experiential_lookup_item과 Experiential_add_item은 커널 전역에서 사용할 수 있도록 EXPORT_SYMBOL을 이용하여 Symbol 테이블에 등록하여 사용한다.

2.3.1 Table내의 경험적 데이터 Update 처리

새로운 경험적 데이터가 extendible hash table에 전송 시, 먼저 extendible hash table내에 같은 process_name의 경험적 데이터를 확인한다. 만약 있을 경우 기존의 경험적 데이터와 새로운 경험적 데이터를 가공하여 저장한다. 이 때 각각의 데이터 적용 수치는 테스트 과정을 통해 결정하였다.

2.3.1.1 time slice

```
int Experiential_add_item(){
  new time slice = (old time slice + new time slice) / 2;
}
```

기존의 경험적 데이터와 새로운 경험적 데이터의 비율을 5:5로 평균을 적용하였다.

2.3.1.2 Frequency

```
int Experiential_add_item(){
  // 스케줄링에서 실행빈도수는 우선순위 가중치 값이 된다.
  if(실행빈도수 < 100)
        실행빈도 수 ++;
}
```

extendible hash table에 경험적 데이터가 입력될 때, 해당 process_name의 경험적 데이터가 존재할 시에 해당 프로세스의 경험적 데이터에서 실행빈도 count를 증가 시키는 방법으로 실행 빈도를 알아냈다. 프로세스의 실행 빈도 값은 Normal Process와 RealTime Process의 경계인 100을 최대 실행 빈도 값으로 하고, 실행 빈도에 따른 비율을 우선순위에 적용하였다.

2.3.1.3 Process Type

```
int Experiential_add_item() {
  if( 해당 프로세스가 커널 내부 특정 함수에 접근 여부 )
        process_type = 해당 flag에 의한 값.
}
```

프로세스 종류를 확인 하는 방식은 커널 내부 함수 scanning을 통해 확인 하였다. 이때 사용한 커널 내부 특정 함수는 inet_create()와 snd_pcm_open() 함수를 사용하였다. inet_create()에 접근한 process는 Network Process로 분류되고 snd_pcm_open()에 접근한 process는 Multimedia Process로 구분하였다. 이렇게 구분된 프로세스는 차후 같은 프로세스의 실행 시, 자신에 맞는 우선순위를 적용하였다.

3. 테스트 결과 및 분석

3.1 측정 Tool : TSS 레지스터

```
static __inline__ unsigned long int using_time(void){
    unsigned long long int x;
    __asm__ __volatile__(".byte 0x0f,0x31" : "=A" (x));
```

```
    return x;
}
```

각 프로세스의 실행 시간을 측정하기 위하여 TSS 레지스터의 값을 위 함수를 이용하여 측정 하였다.

3.2 테스트 환경

- Pentium(R)4 CPU 2.53 GHz , 512MB RAM
- OS : Fedora Core 5 (Kernel : Linux 2.6.18)

3.3 Normal Process에 경험적 Data 적용 결과

Matrix 10000*65000의 곱을 수행하는 프로세스에 경험적 time_slice 데이터와 경험적 frequency 값을 적용하여 10000회 테스트 후 최대, 최소 값 제외하고 결과의 평균을 구하였고 TSS 레지스터를 이용하여 프로세스 실행 시간 측정하였다.

	기존 커널	적용 커널	효율
프로세스 실행 시간 평균 값 (TSS 레지스터 값)	4002342342023	388178822134	3.11%

표 1. 일반 프로세스 테스트 결과표

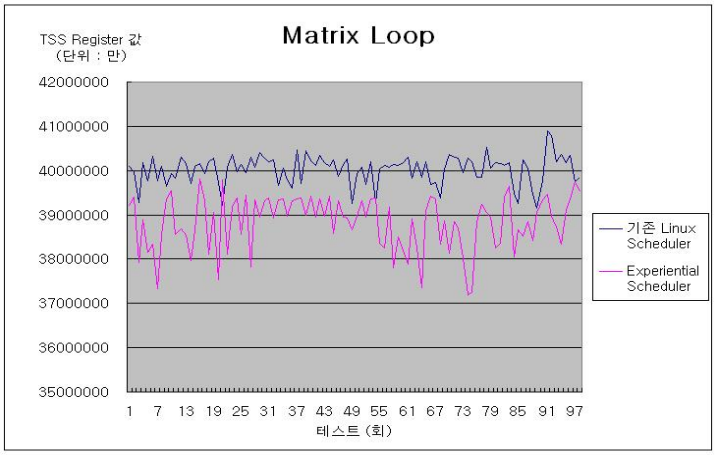

그림 3. 일반 프로세스 테스트 결과 비교 그래프

위의 결과로 확인할 수 있듯이 기존의 리눅스 커널보다 3.11%의 효율을 나타내는 것을 확인 할 수 있다. 10000회의 테스트 중 각 100회씩을 그래프에 도식한 그림 3에서도 Experiential Scheduler를 적용한 커널이 확연하게 아래쪽에 형성되는 것을 확인 할 수 있다. 이외에 수차례의 테스트 결과 현재 OS상에 현재 실행 중인 프로세스의 개수와 종류에 따라 결과 차이를 보였다. 이를 보완하기 위해 기존커널 10000회 테스트 후 Experiential Scheduler가 적용된 Experiential 커널을 10000회를 측정하였고 매 측정 간격은 최소화 시켰다.

3.2 Network Process 경험적 Data 적용 결과

Network Process에 경험적 time_slice 데이터와 경험적 frequency 값과 process_type 값을 적용하고 결과를 테스트하기 위하여 자체 제작한 echo server-client 프로그램에서 15byte 길이의 문자열을 50번 전송한 결과를 1회로 TSS 레지스터 값을 이용하여 client 측에서 측정하였다. 각 측정 간격은 최소화 시켰고 결과 값은 100회 측정 후 최대, 최소 값은 제외하고 평균을 구했으며 결과는 다음과 같다.

	기존 커널	적용 커널	효율
프로세스 실행 시간 평균 값 (TSS 레지스터 값)	5389657	5358856	0.57%

표 2. 네트워크 프로세스 테스트 결과표

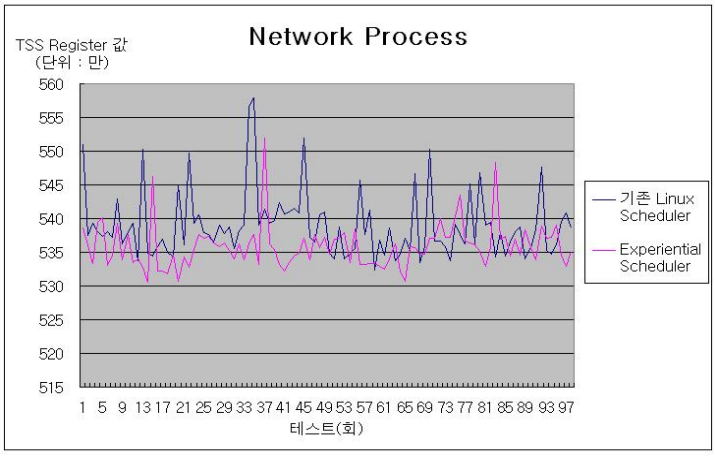

그림 4. 네트워크 프로세스 테스트 결과 비교 그래프

위의 결과에서 효율은 0.57%로 미약하였다. 네트워크 프로세스의 경우에는 프로세스의 실행시간 대부분이 I/O 바운드에 해당하기 때문에, CPU상에서의 프로세스 실행 시간보다 네트워크 상에서의 전송시간이 비중을 많이 차지하기 때문으로 분석된다.

3.2 Multimedia Process 경험적 Data 적용 결과

Multimedia Process에 경험적 time_slice 데이터와 경험적 frequency 값과 process_type 값을 적용한 결과에 대한 테스트는 Audio와 Video로 나누어 테스트 하였다. 먼저 Audio관련 Multimedia Process는 wav확장자를 가지는 노래를 1초간 재생한 결과를 TSS레지스터를 이용하여 측정 하였다. 이때 노래 재생에는 mplayer를 사용하였다. 결과는 100회 측정 후 최대, 최소 값은 제외한 평균이다. 음성파일의 경우에 차이를 보일 수 있는 부분은 mplayer 프로그램의 실행 부분과 종료부분에서 차이가 나므로 1초의 짧은 파일의 재생 시간을 측정 하였다. 측정 결과는 다음과 같다.

	기존 커널	적용 커널	효율
프로세스 실행 시간 평균 값 (TSS 레지스터 값)	3187596506	3186009439	0.05%

표 3. 멀티미디어(음성) 프로세스 적용 결과표

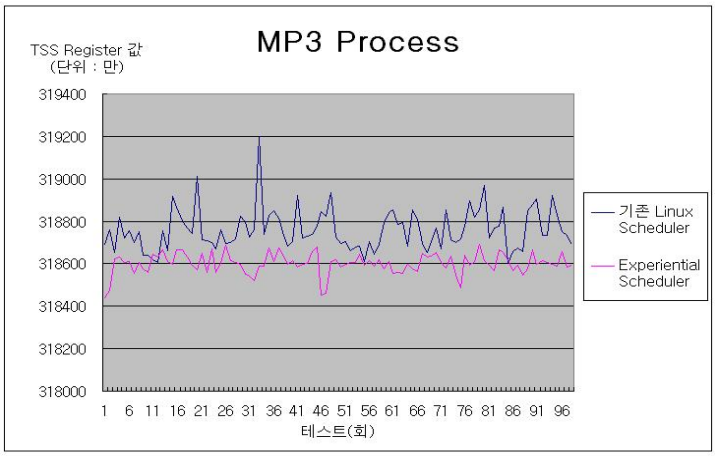

그림 5. 멀티미디어(음성) 프로세스 테스트 결과 비교 그래프

Multimedia Process중 영상관련 프로세스의 적용 테스트를 위해서 음성 관련 프로세스와 마찬가지로 1초의 짧은 영상을 테스트 하였다. asf포멧의 파일을 mplayer에서 재생시켰고, 100회 테스트 후 최대, 최소 값은 제외하였다. 시간 측정은 TSS레지스터를 이용하였다. 결과는 다음과 같다.

	기존 커널	적용 커널	효율
프로세스 실행 시간 평균 값 (TSS 레지스터 값)	3842465767	3817455039	0.66%

표 4. 멀티미디어(영상) 프로세스 적용 결과표

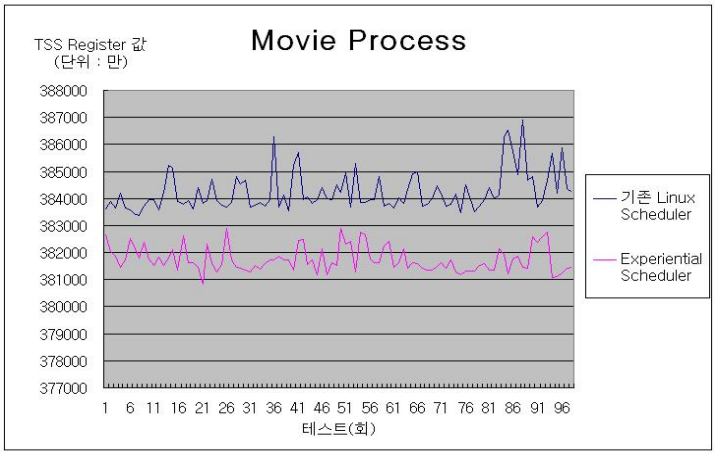

그림 6. 멀티미디어(영상) 프로세스 테스트 결과 비교 그래프

Multimedia Process의 적용 후 일반 프로세스보다 좋은 효율을 기대했지만, 결과는 작은 성능 향상에 그쳤다. 이러한 결과는 멀티미디어, 네트워크 프로세스 모두 I/O bound에 많은 시간을 사용하기 때문으로 판단된다.

4. 결론

Experiential Scheduler는 다양한 분야에 적용 가능하다. 일단 리눅스 기반이라는 점에서 임베디드 시스템으로의 적용이 가능하고, 여타 다른 OS로의 발전가능성도 있다. 하지만, I/O bound 중심의 프로세스의 경우 낮은 효율을 보였고, CPU bound위주의 프로세스에서 보다 좋은 효율을 보였다.

하지만, scheduler라는 OS의 핵심 부분이 수정된다는 점은 그만큼 위험을 안고 있다. 많은 테스트와 결과 확인 및 동작에 이상이 없음을 확인 하였지만, 작은 delay에도 많은 오버헤드를 나타내는 scheduler인 만큼 항상 조심스러운 접근이 필요하다. 이를 극복하기 위해서는 좀더 개선된 자료구조 형태와 어셈블리 레벨에서의 수정을 통해 오버헤드를 극복한다면 더 좋은 성능을 기대 할 수 있다.

만약, RT-OS 혹은 RT-Linux등 real time이 보장되어야 하는 OS에서의 적용을 위해서는 기존의 Experiential Scheduler에서 적용하였던 경험적 데이터 방식에 대한 연구 및 재설계가 필요하다.

5. 참고문헌

[1] Silberschatz, Galvin, Gagne , "OPERATING SYSTEM CONCEPTS 6th" , 152p, 2004 , WILEY

[2] Daniel P. Bovet, “리눅스 커널의 이해”, 401p, 2002, 한빛미디어

[3] AtulNegi/KishoreKumar P , “Applying Machine Learning Techniques to Improve Linux Process Scheduling”, International Conference On High Performance Computing , Department of Computer and Information Sciences University of Hyderabad, INDIA , 2005