운영체제 프로그래밍 과제

< Virtual Memory Management Simulator >

이번 프로그래밍 과제 수업 시간에 배운 Virtual Memory Systems에서 Multilevel Page Table 과 Inverted Page Table system을 구현하여 시뮬레이션 해 보는 것입니다. 제공 되는 mtraces 디렉토리에 실제 프로그램 수행 중 접근한 메모리의 주소(Virtual address)를 순차적으로 모아 놓은 memory trace가 있습니다. 각각의 trace file에 들어 있는 memory trace 포맷은 다음과 같습니다.

```
0041f7a0 R
13f5e2c0 R
05e78900 R
004758a0 R
31348900 W
```

앞의 8문자는 접근된 메모리의 주소를 16진수로 나타낸 것이고(32bits) 그 뒤의 R 또는 W 해당 메모리 주소에 Read를 하는지 Write를 하는지 각각을 나타냅니다. 이 trace는 다음 코드(fscanf())를 사용하여 읽어 들이면 됩니다. (본 과제에서 R/W 는 중요하지 않습니다.)

```
unsigned addr;
char rw;
...
fscanf(file,"%x %c",&addr,&rw);
```

Virtual Memory System simulator에서 virtual address 크기는 32bits 이고 page의 사이즈는 4Kbytes(12bits) 으로 가정 합니다.

0) Virtual Memory Simulator 인자는 다음과 같습니다.

- 첫 번째 (1st level 주소 bits 크기) :

1^{st} level page table에 접근에 사용되는 메모리 주소 bits 의 수. 예를 들어 인자 값이 8 이면 Virtual address 32 bits 중에 앞의 8bits 가 1^{st} level page table의 접근에 사용되며 따라서 1^{st} level page table의 entry의 수는 2^8 이 됩니다. Virtual address 가 32bits 이고 page 사이즈가 4Kbytes(12bits) 로 고정 되어 있기 때문에 첫 번째 인자가 결정되면 자동적으로 2^{nd} level page table에 접근에 사용되는 주소 bits의 수가 결정됩니다. 위의 예에서 2^{nd} level의 주소 bits 수는 32(virtual address bits)-12(page size bits)-8 (1^{st} level 주소 bits 크기)로 계산됩니다.

- 두 번째 (Physical Memory 크기 bits) :

Physical Memory의 크기를 나타내는 인자입니다. 예를 들어 인자 값이 ***n***이면 Physical Memory의 크기는 2^n bytes 가 됩니다. 두 번째 인자의 값으로는 32 보다 더 큰 값을 지원해도 되지만 최소한 12부터 32 bits 까지 반드시 지원 하여야 합니다. 예를 들어 해당 값이 12이면 Physical Memory의 크기는 4Kbytes이고 하나의 Frame으로 구성되었음을 말합니다.

- 세 번째 및 그 이후의 인자들 :

세 번째부터 그 이후의 인자 수는 제한이 없습니다. 이들 인자로는 메모리 trace를 저장한 파일의 이름이 나옵니다. mtraces 디렉토리에는 gcc, bzip, sixpack, swim의 실제 수행 프로그램에

서 얻은 메모리 trace와 random 하게 만들어진 random0, random2 trace 파일이 있고 이들 각각의 파일에는 100만 개의 memory trace가 들어 있습니다.
시뮬레이션에서는 각각의 trace 파일에 있는 메모리 trace는 각각의 다른 프로세서가 수행하며 메모리를 접근한다고 가정합니다. 예를 들어 세 번째 인자로 "gcc.trace bzip.trace" 의 두 파일이 명시되면 process 0는 gcc.trace파일에 있는 메모리 trace의 메모리 주소를 차례대로 접근하게 되고 process 1 은 bzip.trace 파일에 있는 메모리 trace의 메모리 주소를 차례대로 접근한다고 가정 합니다. 따라서 세 번째 인자와 그 이후의 인자의 수만큼의 (명시된 trace 파일의 수) process 가 수행 된다 가정 합니다.
세 번째 인자와 그 이후의 인자로 나오는 trace 파일은 process 0부터 시작하여 process 1, 2, 3 … 식으로 차례대로 할당됩니다. 각 process는 process 0부터 수행하는데 한번 메모리 접근을 하면 다음 process가 수행하여 메모리 접근을 하는 식으로 번갈아 차례대로 메모리에 접근하며 수행된다고 가정합니다.

1) Two-level Page Table System
첫 번째 인자가 결정 되면 각 process 의 1st level page table이 구성 됩니다. 하지만 2nd level page table은 메모리에 접근 하면서 필요한 경우에만 구성하면 됩니다. process 가 memory trace에서 읽어들인 virtual address를 접근하게 되면 수업에서 배운 대로 차례로 1st level page table을 접근하고 그 다음 2nd level page table에 접근 하게 됩니다. 최조의 접근에서는 2nd level page table이 없을 수 있고 이럴 경우에는 해당 page table을 만들어 구성하여야 합니다. 2nd level page table 접근하여 page table entry를 확인하고 해당 page에 이미 physical memory frame 할당 되어 있으면 Page hit 이고 할당 되어 있지 않으면 Page faults 인데 이 경우 page replacement algorithm (LRU) 에 따라 frame을 할당하게 됩니다. 기존에 할당 되어 있던 frame을 다른 page 에 할당 할 경우 기존의 매핑 정보를 invalid하게 설정해야 합니다. frame 이 할당 되어 있거나 할당되면 physical address를 구성할 수 있습니다.

Virtual address의 Physical address의 변환이 끝날 때 마다 다음과 동일한 (인자의 이름은 바꾸어도 됩니다) 포맷의 프린트문으로 변환 정보를 출력합니다.
printf("2Level procID %d traceNumber %d virtual addr %x pysical addr %x\n", i, procTable[i].ntraces, virtualAddress, physicalAddress);

Two-level Page Table을 시뮬레이션 하면서 다음과 같은 정보를 모아 시뮬레이션이 끝나면 출력합니다. 이들 정보는 각 process에 해당 하는 정보입니다. (출력문 참조)

- trace 파일 이름 (procTable[i].traceName)
- 처리한 memory trace 의 수 (procTable[i].ntraces)
- 만들어진 2nd level page table의 수 (procTable[i].num2ndLevelPageTable)
- Page Faults 의 수 (procTable[i].numPageFault)
- Page Hits의 수 (procTable[i].numPageHit)

이들 정보는 다음과 같은 관계가 성립된다.
procTable[i].numPageHit + procTable[i].numPageFault == procTable[i].ntraces

2) Inverted Page Table System
Inverted Page Table 에서는 Hash Table을 사용하여 process의 virtual page 와 frame간의 매핑 정보를 저장합니다. Hash Table의 크기는 frame의 수와 같고 hash 함수로는 다음과 같습

니다.

Hash Table index 값 = (virtual page number + process id) % frame 의 수

process 가 접근하려하는 virtual address에서 virtual page number를 얻어내어 process id 를 더하고 두 번째 인자(Physical Memory의 크기)에서 frame의 수를 얻어서 % 연산으로 Hash table의 index값을 계산합니다.
같은 Hash table의 entry에 여러 개의 매핑 정보가 있을 수 있는데 가장 최근에 만들어진 매핑 정보가 가장 먼저 접근 될 수 있도록 앞쪽에 삽입되어야 합니다. Hash table의 entry에 연결된 매핑 정보를 다 찾아도 접근하고자하는 virtual page number 와 process id를 발견할 수 없으면 Page faults이고 이와 달리 발견하면 해당 매핑 정보에 있는 frame number를 사용하여 physical address를 만들어 내면 됩니다. Page faults 가 발생하면 page replacement algorithm (LRU) 에 따라 frame을 할당하게 됩니다. 기존에 할당 되어 있던 frame을 다른 page 에 할당 할 경우 기존의 매핑 정보를 없애야합니다.

Virtual address의 Physical address의 변환이 끝날 때 마다 다음과 동일한 (인자의 이름은 바꾸어도 됩니다) 포맷의 프린트문으로 변환 정보를 출력합니다.
printf("IHT procID %d traceNumber %d virtual addr %x pysical addr %x\n", i, procTable[i].ntraces,virtualAddress, physicalAddress);

Inverted Page Table을 시뮬레이션 하면서 다음과 같은 정보를 모아 시뮬레이션이 끝나면 출력합니다. 이들 정보는 각 process에 해당 하는 정보입니다. (출력문 참조)

- trace 파일 이름 (procTable[i].traceName)
- 처리한 memory trace 의 수 (procTable[i].ntraces)
- Inverted Hash Table 접근의 수 (procTable[i].numIHTConflictAccess) :
Hash table에서 매핑 정보에 접근한 수를 나타낸다.
- Empty Inverted Hash Table Access (procTable[i].numIHTNULLAccess) :
Hash table에 접근했을 때 해당 엔드리가 연결된 매핑 정보를 하나도 갖지 않은 경우의 수를 나타낸다.
- Non-Empty Inverted Hash Table Access (procTable[i].numIHTNonNULLAcess)
Hash table에 접근했을 때 해당 엔드리가 연결된 매핑 정보를 하나라도 갖는 경우의 수를 나타낸다.
- Page Faults 의 수 (procTable[i].numPageFault)
- Page Hits의 수 (procTable[i].numPageHit)

이들 정보는 다음과 같은 관계가 성립된다.
procTable[i].numPageHit + procTable[i].numPageFault == procTable[i].ntraces
procTable[i].numIHTNULLAccess + procTable[i].numIHTNonNULLAcess == procTable[i].ntraces

3) Page replacement Algorithm (LRU)
이번 과제의 virutal memory management simulation 에서는 Page Replacement Algorithm으로 LRU 방법을 사용한다. 따라서 매번의 Memory 참조를 수행할 때 마다 (매 한 개의 memory 처리) LRU에 따라 다음에 replace될 frame의 순서를 정하여야 한다. Global

replacement를 적용하여 process에 상관없이 모든 frame에 적용된다. LRU의 구현으로는 수업에서 설명하였듯이 linked list 방법이나 counter를 이용한 방법을 사용할 수 있다. Hint로 보여준 프로그램에서는 linked list방식을 사용하고 있다.

4) 주의 사항

memsimhw.c를 사용하여 프로그램을 완성합니다.

0) memsimhw.c를 사용하지만 첫 번째 주석 (학생 이름, 번호등)과 printf 문 (인자이름은 변경가능)을 제외한 모든 것을 바꾸어도 됩니다.
1) 제공된 memsimhw.c 는 참조용입니다. 특히 main() 함수는 완전히 불완전합니다. 참고만 하시기 바랍니다.
2) 함수 내에 있는 ***모든*** printf 문은 모두 반드시 사용합니다. 단 printf 문에 사용되는 인자의 이름은 바꾸어 사용하여도 됩니다. 함수에 사용된 인자의 이름 또한 바꾸어 사용하여도 됩니다.
또 Two-level Page Table System에서는
printf("2Level procID %d traceNumber %d virtual addr %x pysical addr %x\n", i, procTable[i].ntraces, virtualAddress, physicalAddress);

과 Inverted Page Table System 에서는
printf("IHT procID %d traceNumber %d virtual addr %x pysical addr %x\n", i, procTable[i].ntraces,virtualAddress, physicalAddress);
사용하여 주소변환을 출력하는 것도 반드시 포함해야 합니다.

3) 참고로 two-level page table 과 inverted page table의 simulation은 다음과 같이 진행합니다. process 0, process 1, process 2 ,.... process N 에 tracefile 0, tracefile 1, tracefile 2,.... tracefile N 이 각각 할당되었다고 가정합니다(세번 째 인자 참조)

각 process 들이 tracefile 의 trace를 처리하는 순서입니다.

process 0 이 tracefile 0 에 있는 **첫 번째** trace읽고 처리합니다.
process 1 이 tracefile 1 에 있는 첫 번째 trace읽고 처리합니다.
.....
process N 이 tracefile N 에 있는 첫 번째 trace읽고 처리합니다.

process 0 이 tracefile 0 에 있는 **두 번째** trace읽고 처리합니다.
process 1 이 tracefile 1 에 있는 두 번째 trace읽고 처리합니다.
.....
process N 이 tracefile N 에 있는 두 번째 trace읽고 처리합니다.
.....
.....
process 0 이 tracefile 0 에 있는 **마지막** trace읽고 처리합니다.
process 1 이 tracefile 1 에 있는 마지막 trace읽고 처리합니다.

.....
process N 이 tracefile N 에 있는 마지막 trace읽고 처리합니다.

즉 각 process 가 할당 된 tracefile의 모든 trace를 처리하면 끝나게 됩니다.

“처리합니다”의 처리과정은 two-level page 또는 inverted page table의 virtual address translation 과정의 수행을 의미하며 two-level page의 simulation이 다 끝난 후 inverted page table의 translation 과정을 simulation 하는 방식으로 합니다.

<memsimhw.c>

```
//
// Virual Memory Simulator Homework
// Two-level page table system
// Inverted page table with a hashing system
// Student Name:
// Student Number:
//
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define PAGESIZEBITS 12                 // page size = 4Kbytes
#define VIRTUALADDRBITS 32              // virtual address space size = 4Gbytes

struct pageTableEntry {
        int level;                      // page table level (1 or 2)
        char valid;
        struct pageTableEntry *secondLevelPageTable;    // valid if this entry is for the first level
page table (level = 1)
        int frameNumber;                                                        // valid if this
entry is for the second level page table (level = 2)
};
struct framePage {
        int number;                     // frame number
        int pid;                    // Process id that owns the frame
        int virtualPageNumber;                  // virtual page number using the frame
        struct framePage *lruLeft; // for LRU circular doubly linked list
        struct framePage *lruRight; // for LRU circular doubly linked list
};
struct invertedPageTableEntry {
        int pid;                                // process id
        int virtualPageNumber;          // virtual page number
        int frameNumber;                // frame number allocated
        struct invertedPageTableEntry *next;
};
struct procEntry {
        char *traceName;                // the memory trace name
        int pid;                                // process (trace) id
        int ntraces;                            // the number of memory traces
        int num2ndLevelPageTable;       // The 2nd level page created(allocated);
```

```
        int numIHTConflictAccess;       // The number of Inverted Hash Table Conflict Accesses
        int numIHTNULLAccess;           // The number of Empty Inverted Hash Table Accesses
        int numIHTNonNULLAcess;                 // The number of Non Empty Inverted Hash Table
Accesses
        int numPageFault;                       // The number of page faults
        int numPageHit;                         // The number of page hits
        struct pageTableEntry *firstLevelPageTable;
        FILE *tracefp;
};

struct framePage *oldestFrame; // the oldest frame pointer
int firstLevelBits, phyMemSizeBits, numProcess;

void initPhyMem(struct framePage *phyMem, int nFrame) {
        int i;
        for(i = 0; i < nFrame; i++) {
                phyMem[i].number = i;
                phyMem[i].pid = -1;
                phyMem[i].virtualPageNumber = -1;
                phyMem[i].lruLeft = &phyMem[(i-1+nFrame) % nFrame];
                phyMem[i].lruRight = &phyMem[(i+1+nFrame) % nFrame];
        }
        oldestFrame = &phyMem[0];
}

void secondLevelVMSim(struct procEntry *procTable, struct framePage *phyMemFrames) {
                for(i=0; i < numProcess; i++) {
                printf("**** %s *****\n",procTable[i].traceName);
                printf("Proc %d Num of traces %d\n",i,procTable[i].ntraces);
                printf("Proc %d Num of second level page tables allocated
%d\n",i,procTable[i].num2ndLevelPageTable);
                printf("Proc %d Num of Page Faults %d\n",i,procTable[i].numPageFault);
                printf("Proc %d Num of Page Hit %d\n",i,procTable[i].numPageHit);
                assert(procTable[i].numPageHit + procTable[i].numPageFault == procTable[i].ntraces);
        }
}

void invertedPageVMSim(struct procEntry *procTable, struct framePage *phyMemFrames, int nFrame) {
        for(i=0; i < numProcess; i++) {
                printf("**** %s *****\n",procTable[i].traceName);
                printf("Proc %d Num of traces %d\n",i,procTable[i].ntraces);
                printf("Proc %d Num of Inverted Hash Table Access Conflicts
%d\n",i,procTable[i].numIHTConflictAccess);
                printf("Proc %d Num of Empty Inverted Hash Table Access
%d\n",i,procTable[i].numIHTNULLAccess);
                printf("Proc %d Num of Non-Empty Inverted Hash Table Access
%d\n",i,procTable[i].numIHTNonNULLAcess);
                printf("Proc %d Num of Page Faults %d\n",i,procTable[i].numPageFault);
                printf("Proc %d Num of Page Hit %d\n",i,procTable[i].numPageHit);
                assert(procTable[i].numPageHit + procTable[i].numPageFault == procTable[i].ntraces);
                assert(procTable[i].numIHTNULLAccess + procTable[i].numIHTNonNULLAcess ==
procTable[i].ntraces);
        }
}
```

```
int main(int argc, char *argv[]) {
        int i;
        if (argc < 4) {
            printf("Usage : %s firstLevelBits PhysicalMemorySizeBits TraceFileNames\n",argv[0]); exit(1);
        }
        if (phyMemSizeBits < PAGESIZEBITS) {
                printf("PhysicalMemorySizeBits %d should be larger than PageSizeBits
%d\n",phyMemSizeBits,PAGESIZEBITS); exit(1);
        }
        if (VIRTUALADDRBITS - PAGESIZEBITS - firstLevelBits <= 0 ) {
                printf("firstLevelBits %d is too Big\n",firstLevelBits); exit(1);
        }
                // initialize procTable for two-level page table
        for(i = 0; i < numProcess; i++) {
                // opening a tracefile for the process
                printf("process %d opening %s\n",i,argv[i+3]);
        }
        int nFrame = (1<<(phyMemSizeBits-PAGESIZEBITS)); assert(nFrame>0);
        printf("\nNum of Frames %d Physical Memory Size %ld bytes\n",nFrame, (1L<<phyMemSizeBits));
                printf("============================================================\n");
        printf("The 2nd Level Page Table Memory Simulation Starts .....\n");
        printf("============================================================\n");
                // initialize procTable for the inverted Page Table
        for(i = 0; i < numProcess; i++) {
                // rewind tracefiles
                rewind(procTable[i].tracefp);
        }
        printf("============================================================\n");
        printf("The Inverted Page Table Memory Simulation Starts .....\n");
        printf("============================================================\n");
                return(0);
}
```

*) 수행 예

```
$ ./memsim 10 32 ../mtraces/gcc.trace ../mtraces/bzip.trace
process 0 opening ../mtraces/gcc.trace
process 1 opening ../mtraces/bzip.trace

Num of Frames 1048576 Physical Memory Size 4294967296 bytes
============================================================
The 2nd Level Page Table Memory Simulation Starts .....
============================================================
2Level procID 0 traceNumber 1 virtual addr 2f8773d8 pysical addr 3d8
2Level procID 1 traceNumber 1 virtual addr 6645b58 pysical addr 1b58
2Level procID 0 traceNumber 2 virtual addr 3d729358 pysical addr 2358
2Level procID 1 traceNumber 2 virtual addr 6645b58 pysical addr 1b58
......
......
2Level procID 0 traceNumber 999999 virtual addr 2f8773e0 pysical addr 3ae3e0
2Level procID 1 traceNumber 999999 virtual addr 6645ba0 pysical addr 723ba0
```

```
2Level procID 0 traceNumber 1000000 virtual addr 3d729358 pysical addr 24358
2Level procID 1 traceNumber 1000000 virtual addr 5fe5180 pysical addr 2eb180
**** ../mtraces/gcc.trace *****
Proc 0 Num of traces 1000000
Proc 0 Num of second level page tables allocated 164
Proc 0 Num of Page Faults 2852
Proc 0 Num of Page Hit 997148
**** ../mtraces/bzip.trace *****
Proc 1 Num of traces 1000000
Proc 1 Num of second level page tables allocated 39
Proc 1 Num of Page Faults 317
Proc 1 Num of Page Hit 999683
============================================================
The Inverted Page Table Memory Simulation Starts .....
============================================================
IHT procID 0 traceNumber 1 virtual addr 2f8773d8 pysical addr 3d8
IHT procID 1 traceNumber 1 virtual addr 6645b58 pysical addr 1b58
IHT procID 0 traceNumber 2 virtual addr 3d729358 pysical addr 2358
IHT procID 1 traceNumber 2 virtual addr 6645b58 pysical addr 1b58
....
....
IHT procID 0 traceNumber 999999 virtual addr 2f8773e0 pysical addr 3ae3e0
IHT procID 1 traceNumber 999999 virtual addr 6645ba0 pysical addr 723ba0
IHT procID 0 traceNumber 1000000 virtual addr 3d729358 pysical addr 24358
IHT procID 1 traceNumber 1000000 virtual addr 5fe5180 pysical addr 2eb180
**** ../mtraces/gcc.trace *****
Proc 0 Num of traces 1000000
Proc 0 Num of Inverted Hash Table Access Conflicts 997160
Proc 0 Num of Empty Inverted Hash Table Access 2840
Proc 0 Num of Non-Empty Inverted Hash Table Access 997160
Proc 0 Num of Page Faults 2852
Proc 0 Num of Page Hit 997148
**** ../mtraces/bzip.trace *****
Proc 1 Num of traces 1000000
Proc 1 Num of Inverted Hash Table Access Conflicts 1001424
Proc 1 Num of Empty Inverted Hash Table Access 316
Proc 1 Num of Non-Empty Inverted Hash Table Access 999684
Proc 1 Num of Page Faults 317
Proc 1 Num of Page Hit 999683
```