운영체제 3차 과제

- Process scheduler, virtual memory and memory swapping simulation -

목차

2014147550
컴퓨터과학과
강효림

*참고문헌은 없어서 생략하였음

1. 프로젝트 개요

OS가 수행하는 프로세스 스케줄링 및 가상메모리 관리(physical memory에서의 스와핑 포함)를 시뮬레이션하는 c 프로그램을 작성한다.

2. 프로그램 구조와 스케줄러, 메모리 관리의 구현방법

가. 구성 소스 파일

mydatastruct.h mydatastruct.c simulator.c로 구성되고, 자체 제작 자료구조와 함수는 모두 mydatasturct.h에 선언, mydatastruct.c에 구현되어 있고, simulator.c에는 이를 활용하는 main 함수만이 존재한다.

나. 프로그램 개관(simulator.c를 중심으로)

input에서 입력값을 받고, input, output file의 stream을 여는 일, 그리고 입력값에 맞게 변수를 할당해주는 일은 제외하고는, 대부분의 task는 while(1) 루프 안에서 실행된다.

가장 먼저 feedCnt(feedSize만큼의 cpu quantum을 할당해줄 시기를 정하기 위한 변수), cpuCycle을 증가시킨다. 그리고 eventFetch함수를 통하여 input파일에서 이벤트를 가져온다. 이 때 당해 cpuCycle에 해당하는 event가 없다면 eventFetch함수는 NULL을 반환하고, 있다면 그 이벤트를 표현하는 문자열을 반환한다. 그리고 이를 받은 fetchedEvent를 detEventType함수에 넣어 이벤트의 성질을 파악하여, 이를 정수로서 eventType에 저장한다. (0: no event, 1: process creation, 2: INPUT)

그 후 feedCnt가 feedFreq와 같다면, 전체 process에 cpu time을 distribute한다. 이 때 전체 프로세스에 cpu time distribution을 위해서 all process queue(aq), 즉 모든 프로세스의 포인터를 갖고 있는 연결 리스트를 이용한다. 이후 sqCtrl 함수로 하여금 sleepQueue(sq)에 있는 프로세스들을 관리한다. 즉 당해 cpuCycle에 sleep이 풀리는 process들이 있다면, 이를 sq에서 제거하여 rq(runQueue)에 enqueue해 주는 작업을 시행한다.

이후 eventType을 확인하여 process creation이면, fetchedEvent를 파싱하여 process code를 추출하고, 이를 바탕으로 새롭게 process를 만들어 rq에 삽입한다. 한편, INPUT event이면, 마찬가지로 fetchedEvent를 파싱하여 input event가 들어온 pid를 확인하고, 이를 IOCtrl함수에 넣어 ioq에서 당해 process를 제거하고 이를 rq에 enqueue한다.

그 후 cpuOccupied변수가 0으로써 cpu가 현재 점유되어있지 않고, rq가 비어있지 않다면 process를 rq에서 가져온다. 이 때 주어진 스펙에서와 같이 rq가 비어있지 않지만 모든 rq 내의 process가 time quantum을 소진해서 process가 fetch될 수 없다면 다시 cpu time을 distribute하고 fetch한다. 이후 fetchedProcess에서 실행할 명령어를 가져온 뒤 이를 system.txt에 출력하고, pExec함수를 통해 실제로 실행한다. 명령허 실행 후 bust되었는지, sleep에 들어갔는지를 확인하여 적절한 처리를 해 준 뒤, while(1)문의 처음으로 돌아간다. 여기서 **종료 조건**은 “totalEventNum == eventCnt, 즉 input의 명령어 개수와 실행된 명렁어의 개수가 같고, aq를 제외한 모든 프로세스 큐가 비어있으며, fetchedProcess가 없을 경우” 이다.

다. 구체적인 구현

1) 프로세스 스케줄링

언급했다시피 4개의 queue(그러나 반드시 front의 원소만 제거된다는 보장이 없는 sq와 ioq는 임의의 자리에서 제거가 가능하게 구현한 자체 자료구조 사용)로써 구현한다. 한편 이러한 queue들은 process_ptr, process_wrapper_ptr를 갖는 process_wrapper로 구성되며 linked list의 형식으로 구현된다. 따라서 예컨대 어떤 process_wrapper_ptr pwp로부터 그 wrapper가 갖는 process의 pid에 접근하고 싶다면 pwp->p->pid로서 접근한다.

모든 프로세스들은 생성되자마자 aq, 즉 all process queue에 삽입된다. 앞으로 여러 task를 수행함에 있어 현재 존재하는 전체 프로세스 중 pid로 특정 프로세스에 접근하고자 할 때, 또는 전체 프로세스 모두에 접근하고자 할 때 본 queue를 사용한다.

rq, ioq, sq는 input의 이벤트와 각 프로세스 code의 event에 따라 앞에서 설명한 바와 같이 적절히 관리된다.

aq	process_wrapper	…	process_wrapper	process_wrapper
rq	process_wrapper	…	process_wrapper	process_wrapper
sq	process_wrapper	…	process_wrapper	process_wrapper
ioq	process_wrapper	…	process_wrapper	process_wrapper

<그림 1> 스케줄링을 위한 큐들

2) 메모리 관리 - 자체 자료구조인 MTREE

(physical)메모리 관리는 자체 자료구조인 MTREE를 이용하여 구현한다. physical memory는 배열로서 존재하지 않고, MTREE로 이루어진 이진 트리에 추상적으로 존재한다. 이하는 MTREE의 자료구조이다.

```
11 typedef struct MTREE* MTREE_PTR;
12
13 typedef struct MTREE{
14     int leaf;
15     int isLeft;
16     int isAlloc;
17     int startIndex;
18     int endIndex;
19     int pid;
20     int allocID;
21     MTREE_PTR parent;
22     MTREE_PTR lc;
23     MTREE_PTR rc;
24 } MTREE;
```

<그림 2> MTREE의 선언

leaf는 당해 MTREE가 leaf인지 여부, isLeft는 당해 MTREE가 parent의 왼쪽 자식인지 여부, isAlloc은 그 MTREE가 표시하는 physical memory가 allocation 되어 있는지 여부, startIndex, endIndex는 그 MTREE가 표시하는 physical memory의 index, pid, allocID는 만일 isAlloc == 1이라면 어떤 프로세스의 어떤 allocID로서 점유되어있는지를 나타내는 정보이다. 그리고 parent는 부모 MTREE의 포인터, lc, rc의 자식 MTREE의 포인터를 나타낸다.

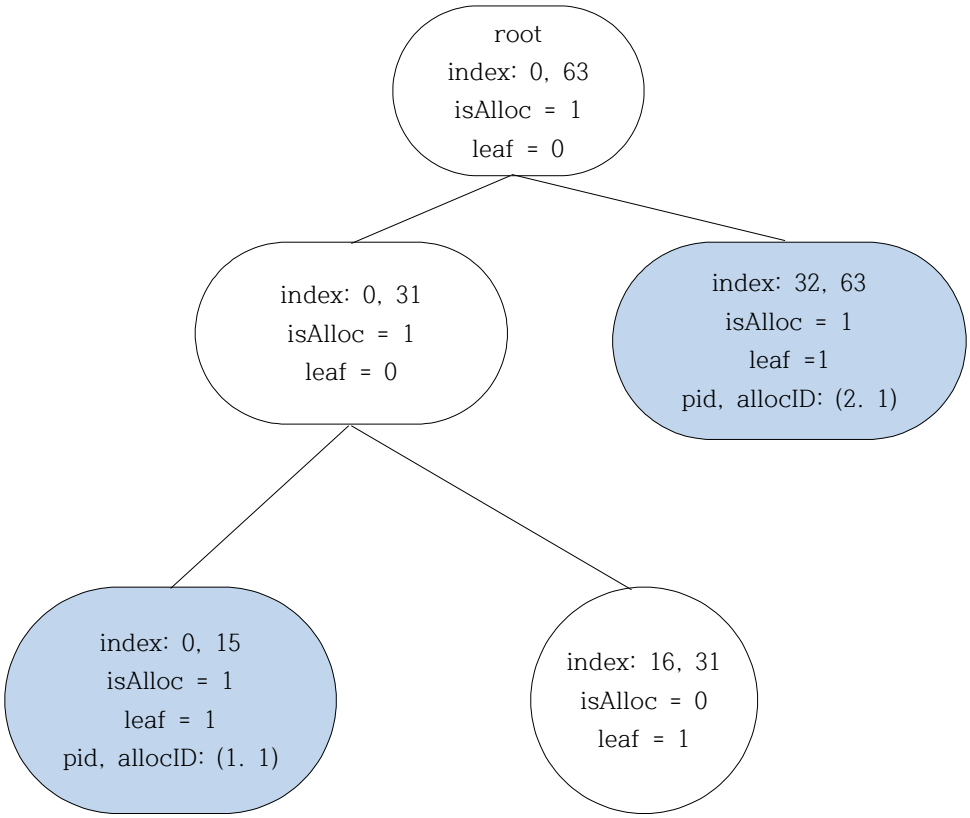
3) physical memory의 allocation

physical memory에 새로운 메모리 할당 명령이나 page fault로 인하여 새롭게 메모리 할당이 일어날 때 어느 MTREE에 할당이 될지 결정하는 함수의 code를 본다. 본 함수는 할당의 목적이 되는 MTREE 구조체의 포인터를 반환한다.

```
521 MTREE_PTR searchForAlloc(MTREE_PTR parent, MTREE_PTR mp, int isLeft, int
522         requestSize, int cpuCycle, int pid, int allocID, FILE* memory){
523     int pSize;
524     if (parent){
525         pSize = parent->endIndex - parent->startIndex + 1;
526     }
527     int mySize = (mp->endIndex) - (mp->startIndex) + 1;
528     int cSize = mySize/2;
529     if (mp->leaf){//if mp is leaf node
530         if ((requestSize > cSize) && (requestSize <= mySize)){
531             if (mp->isAlloc){//if allocated, stop search
532                 return NULL;
533             } else {//found fitting block
534                 fprintf(memory, "%d\t%d\t%d\t%d\n", cpuCycle, pid, allocID, 0);
535                 return mp;
536             }
537         } else if (requestSize <= cSize && !mp->isAlloc){//continue searching
538             mp->leaf = 0;
539             mp->isAlloc = 1;
540             mp->rc = createMTREE(mp, mp->startIndex, mp->endIndex, 0);//spliting block
541             mp->lc = createMTREE(mp, mp->startIndex, mp->endIndex, 1);//and making children
542             return searchForAlloc(mp, mp->lc, 1, requestSize, cpuCycle, pid, allocID, memory);//continue searcing
543         } else {//if allocated stop search
544             return NULL;
545         }
546     } else {//not a leaf node, continue searching
547         MTREE_PTR lc, rc;
548         lc = searchForAlloc(mp, mp->lc, 1, requestSize, cpuCycle, pid, allocID, memory);
549         if (lc){//leftmost first
550             return lc;
551         }
552         rc = searchForAlloc(mp, mp->rc, 0, requestSize, cpuCycle, pid, allocID, memory);
553         return rc;
554     }
555 }
```

<그림 3> searchForAlloc 함수 code

특징적인 부분으로는 leaf에서 requestSize <= cSize && !mp->isAlloc일 때 mp의 양 자식에 새로운 MTREE 구조체를 만들어 할당하고(memory block split으로써, createMTREE 함수 사용) 왼쪽 자식인 lc에 재귀적으로 본 함수를 호출하는 것을 꼽을 수 있다. 따라서 다음과 같은 예시 트리와 같은 형태로 physical memory가 존재하게 된다. physical memory의 크기는 64, pid 1, 2인 process에서 allocID 1로써 크기 16, 32로 두 번의 메모리 할당이 이루어진 경우를 가정하였다. **leaf == 1인 부분이 실제로 physical memory를 추상화하는 부분**이며, **색이 입혀진 부분은 leaf 노드이면서, 실제로 memory가 할당된 부분**이다. 그리고 당해 MTREE가 나타내는 physical memory의 순차적인 표현 또한 그림으로 나타냈다. (다만 프로그램에선 physical memory를 나타내는 별도의 배열은 존재하지 않음)



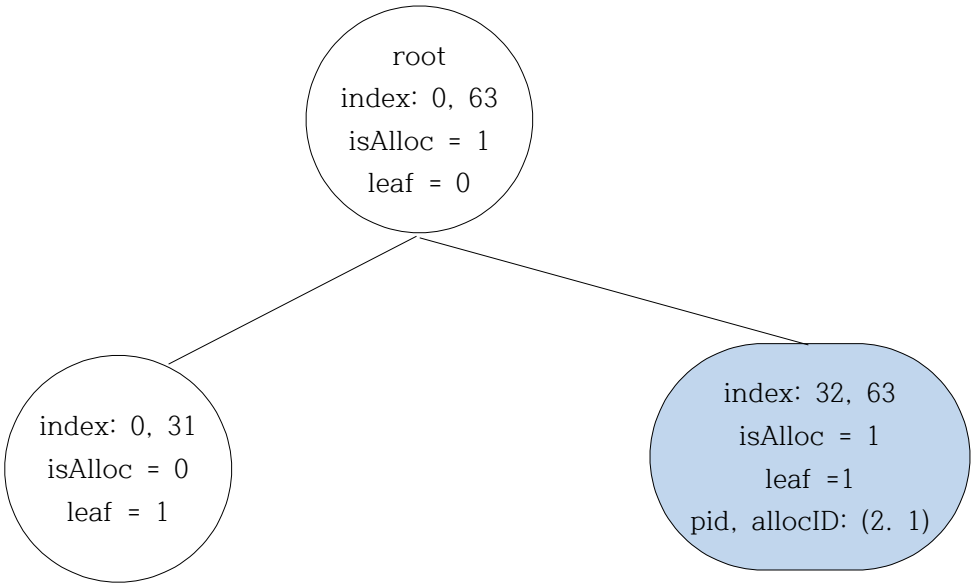
<그림 4> MTREE 예시

0~15	16~31	32~63

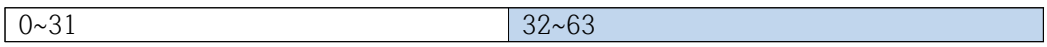
<그림 5> 상기 MTREE가 나타내는 physical memory의 배열 표현

4) physical memory의 free

searchForAccess 함수로 MTREE에서 해제할 pid, allocID를 갖는 MTREE_PTR를 반환받아, memFree 함수로 free작업을 시행한다. 이 때 memFree함수는 memoryMerge 작업을 수반한다. memoryMerge 작업은 대상 MTREE 구조체와 sibling MTREE 구조체의 isAlloc을 확인하여, 둘다 0이라면 두 MTREE 구조체를 free하고 그 parent를 반환하면서 재귀적으로 수행한다. 이는 physical memory에서 buddy system에 의한 block merge의 추상화이다. 이하는 위에 제시한 예시 트리에서 pid, allocID가 각각 1, 1인 메모리를 free 한 뒤 한번의 memoryMerge가 일어난 후의 모습을 나타낸 트리와 physical memory의 순차 표현이다.



<그림 6> pid, allocID가 (1, 1)인 메모리를 free한 이후의 MTREE



<그림 7> 위 MTREE가 나타내는 physical memory 순차표현

5) virtual memory 관리

virtual memory는 process 별로 일차원 배열로 관리되며, 할당은 왼쪽을 기준으로 first fit에 된다. 특기할 점은 없으며, 다만 allocation space를 찾을 때 $O(n)$이 되도록 하였다. physical memory와의 동기화는 pExec 함수 내에서 일어난다. 코드는 다음 페이지에 첨부한다.

6) LRU 구현

process의 포인터와 allocID, 전 원소와 후 원소의 포인터를 갖는 LRU_STRUCTURE로 구성되는 임의 원소 접근이 가능한 큐(이중 연결 리스트로 구현)인 lq로 구현된다. 메모리 접근이나 할당이 일어나면 lq를 조회하고, lq에 당해 PID, allocID를 갖는 LRU_STRUCTURE가 존재한다면 이를 delete한 뒤 lq에 enqueue한다. 없다면 새로 LRU_STRUCTURE를 만들어 enqueue한다(젠가 큐). 메모리가 부족하거나, free 명령이 와서 victimize할 때에는 lq에서 dequeue한 LRU_STRUCTURE의 process, allocID를 갖는 memory를 searchForAccess 함수로 찾아 free한다. 이하는 LRU를 구성하는 자료구조의 선언 부분이다.

```
714 void vmemAlloc(VM_PTR vm, int vmSize, int requireSize, int allocID){
715     int tmp;
716     if (!vm){
717         return;
718     }
719     for(int i=0;i<vmSize;){;
720         tmp = i+1;
721         int j;
722         for (j=i;j<requireSize+i;j++){
723             if (vm[j].allocID != -1){
724                 tmp = j;
725                 break;
726             }
727         }
728         if (j==requireSize+i){
729             for (int k=i;k<requireSize+i;k++){
730                 vm[k].allocID = allocID;
731                 vm[k].valid = 1;
732             }
733             return;
734         }
735         if (tmp < i+1){
736             i += 1;
737         } else {
738             i = tmp;
739         }
740     }
741     printf("  >>vmemAlloc fail<<  \n");
742     return;
743 }
744
745 void vmemFree(VM_PTR vm, int vmSize, int allocID){
746     if (!vm){
747         return;
748     }
749     for (int i=0;i<vmSize;i++){
750         if (vm[i].allocID == allocID){
751             vm[i].allocID = -1;
752             vm[i].valid = 0;
753         }
754     }
755 }
756
```

<그림 8> virtual memory 관리 code

```
62 typedef struct LRU_STRUCTURE* LRU_STRUCTURE_PTR;
63
64 typedef struct LRU_STRUCTURE {
65     PROCESS_PTR p;
66     int allocID;
67     LRU_STRUCTURE_PTR next;
68     LRU_STRUCTURE_PTR prev;
69 } LRU_STRUCTURE;
70
71 typedef struct {
72     int num;
73     LRU_STRUCTURE_PTR front;
74     LRU_STRUCTURE_PTR rear;
75 } LRUQ;
```

<그림 9> LRU를 구현하는 자료구조

3. 개발환경(uname -a의 출력)

```
root@ubuntu:~/OSA2# uname -a
Linux ubuntu 4.4.0-127-generic #153-Ubuntu SMP Sat May 19 10:58:46 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux
```

<그림 10> 개발환경 확인을 위한 uname -a 명령의 출력화면

*editor는 vi, compiler는 gcc를 사용하였다.

4. 수행 과정 중 문제와 해결

가. 자료구조의 문제

C로 코딩을 하였기 때문에, 본 프로젝트를 수행함에 있어 필요한 모든 자료구조(이중연결 리스트로 구현되는 임의 삭제가 가능하면서도 enqueue, dequeue 기능을 지원하는 큐, merge가 재귀적으로 일어나는 이진 트리 등)를 직접 구현해야 했다. 많은 시간을 본 자료구조를 코딩하는 데에 썼던 것 같다.

나. CPU time distribution의 문제

과제 스펙 상 오해가 있어, feedSize만큼의 CPU 시간을 process에 배정해 주는 것에서 애로사항이 있었다. feedSize만큼의 시간으로 "갱신", 즉 만일 cpu time distribution이 일어나는 시기에 어떤 프로세스에 남은 cpu time이 2, feedSize가 10이라면 distribution 후에 그 프로세스의 cpu time이 10이 되는 줄 알았는데, "더해주는" 것이 옳았다. 즉 12가 되었어야 했다. 다만 문제 해결은 코드 2줄 정도를 고치는 정도에서 끝나서 시간을 쓰진 않았다.

다. Buddy system의 구현 문제

원래는 직관적으로 일차원 배열로 구현하고자 하였는데, merge나 split을 구현할 방법이 막막하였다. 따라서 이진 트리로 하여금 meta_memory structure를 구현하고 이를 1차원 배열에 사상시키는 방식으로 하려고 생각하였으나, 본 과제 수행에서 반드시 physical memory가 1차원 메모리 형태로 존재할 필요는 없고, **일차원 배열의 형태로 print 시 당해 meta_memory structure인 이진 트리를 후위순회 하면서 leaf 노드만 출력하는 방식을 활용**함으로써 1차원 메모리가 있는 것처럼 출력할 수 있음을 깨닫게 되었다. 따라서 MTREE 구조체로 구성되는 meta_memory structure를 physical memory 그 자체로 보고 따로 physical memory를 나타내기 위한 1차원 배열을 사용하지 않았다.

5. 결론

OS의 round robin scheduling과 io wait, sleep, 그리고 page swapping까지 포함하는 virtual memory, physical memory management를 실제로 (비록 application level이지만) 구현해 보면서 추상적인 이론에 그치지 않고, 보다 실제적인 OS 구현에 대하여 깊은 이해를 할 수 있었던 것 같다. C언어로 코딩하였고, stdio.h, stlib.h 등 기본 라이브러리를 위주로 사용하여 1000줄을 상회하는 코드를 작성하며 여러 복잡한 기능들을 구현하면서 C를 다루는, 특히 자체적인 자료구조 설계와 포인터를 다루는 능력 또한 향상되었던 것 같다.

6. 부록 - 예시 input에 대한 결과 검증

<그림 11> 결과 검증 스크린샷, example 폴더 내에 예시 파일이 존재