**Multicore Programming Project 3**

담당 교수 : 박성용

이름 : 백상욱

학번 : 20190388

1. **개발 목표**동적 메모리 할당에 사용되는 allocator알고리즘을 구현하고 이에 사용되는 malloc, free, realloc함수를 직접 구현한다. 이때 구현한 allocator에서 높은 throughput과 memory utilization을 높이는 것을 목적으로 한다.
2. **개발 범위 및 내용**
   1. **개발 범위**
3. 동적 메모리 할당을 위한 implicit list, explicit list, segregate list 중 implicit list 방식을 채택하여 구현한다. 그리고 이를 이용하여 malloc, free, realloc을 구현하기 위해서 다른 함수들을 함께 구현한다.
   1. **개발 내용 및 개발 방법**

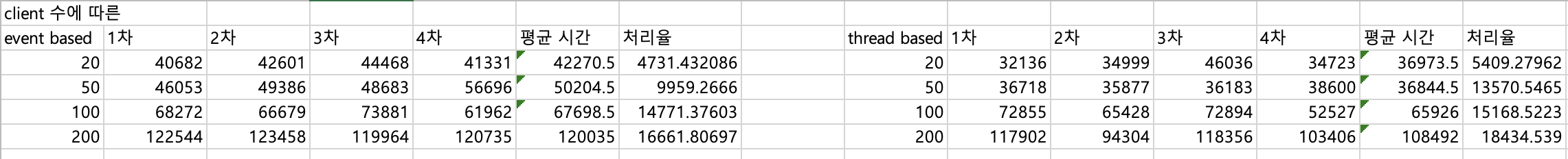
* **Macro :** 우선 다양한 포인터 연산과 워드 단위 연산을 하기 위해서 macro를 정의해둔다. 다만 교과서에 나온 내용과 동일한 매크로는 설명을 생략한다.   
  ALIGNMENT는 double word align을 맞춰야하므로 8로잡는다. 그리고 이를 이용한 ALIGN이라는 매크로를 정의하는데, 이는 특정 사이즈를 줬을 때 align된 사이즈를 반환해주는 매크로로 (((size) + (ALIGNMENT-1)) & ~0x7)와 같이 구현한다.SIZE\_T\_SIZE라는 매크로는 ALIGN과 유사하게 size를 align되어서 반환하는 매크로다. 그리고 Explicit 리스트에서 prev와 next block에 대한 정보를 저장하는 블럭을 가리키는 매크로 NEXT\_EX과 PREV\_EX를 정의한다. PRE\_EX는 GET(HDRP(bp) + WSIZE) PREV\_EX는 GET(HDRP(bp) + DSIZE)로 정의한다.
* static variable : 스태틱 변수를 한가지 추가한 것이 있는데 free\_listp이다. 이는 free\_list를 가리키는 포인터 역할을 하게 된다.
* **mm\_init 함수 :** mm\_init은 초기 힙상태를 초기화하는 함수인데, 기존의 mm\_init과 거의 유사하지만 explicit의 구현을 위해서 초반에 4개의 block이 아닌 6개의 블럭을 할당했 다. 그리고 2개를 explicit 리스트의 구현을 위해서 사용했는데, prologue header, next,prev prologue footer, expilogue로 사용하게 된다. 그리고 free\_listp를 heap\_listp +double word 로 설정한다. 이로서 free\_list의 시작을 구현할 수 있게 되었다.
* **mm\_malloc :** explicit 리스트에 대한 관리는 find\_fit, coalsce, place에서 구현하고 말록 함수의 경우 기존 교과서에 나온 코드와 동일한 코드를 사용했다.
* mm\_free : mm\_free함수의 malloc과 마찬가지로 마찬가지로 교과서 코드를 사용했다. 헤더와 푸터의 alloc에 0을 할당하고 다른 블럭과 합칠 수 있는지 확인하기 위해 coalesce를 호출한다.
* **mm\_realloc :** 크기가 0이라면 free를 의미하는 것이므로 ptr을 free에 넣어준다. ptr이 NULL이라면 단순한 malloc을 의미하는 것이기 때문에 size만큼 malloc을 수행해서 포인터를 반환한다. 그리고 만약 할당해야하는 크기가 현재 할당된 크기보다 작다면 그냥 기존의 포인터를 반환한다. 이 외의 경우에 대해서 새롭게 할당해야하는데, 최적화를 위해서 현재 블럭의 다음 블럭을 확인한다.만약 다음 블럭이 할당되어있지 않다면 그리고 해당 블럭까지의 크기를 현재 블럭의 크기에 더했을 때, 할당하려는 크기보다 커서 할당이 가능하다면 다음블럭을 합쳐서 거기에 할당한다. 이렇게 하면 새롭게 할당하는 것보다 메모리 utilization을 높일 수 있는 것을 실험을 통해 확인했다. 이 외의 경우에는 그냥 새롭게 메모리를 할당한 후에, 새로운 포인터로 기존의 내용을 memcpy로 복사한 후 기존 포인터는 free하고 새로운 포인터를 반환한다.
* extend\_heap() : 교과서에서 사용한 함수와 동일하다. malloc과 realloc이 fit을 찾기 못하고 추가 공간을 필요로하면 그때 호출한다.
* place() : place 함수는 malloc에서 호출되며 asize만큼의 블록을 free에서 찾아내어 할당하는 함수다. 해당 크키만큼 할당했을 때의 블럭 사이즈인 csize를 계산한한ㄷ. 그리고 할당했을 때, 남아있는 블럭의 크기를 rsize로 저장한다. 할당하고 남은 부분은 split해서 저장하기 위함이다. 이때 경우는 3가지가 존재하는데, 첫 번째, 남는 공간이 minimal block size인 4word보다 작은 경우, 이때는 남는 공간을 할당하는게 무의미하기 split하지 않는다. 두번째, 세번째 경우를 고려하기 위해서 기준이 되는 크기를 지정해야한다. explicit과 관련된 자료를 찾아봤을 때, explicit의 메모리 util을 높이지만, 너무 느리지는 않게 높이는 방법으로 이를 선택하는데 가령 50byte를 기준으로 잡는다면, 할당하려는 블럭의 크기가 50바이트 이상인지 이하인지 판단하여 이보다 크다면 risze만큼의 free block을 앞에 배치하고, 이보다 작다면, free block을 뒤로 할당한다. 다양하게 시도해본 결과 100바이트 정도에서 성능이 가장 높게 나오는 것 같아 구현에서는 100을 기준으로 삼았다. 그래서 100을 기준으로 판단하고 free block에 대해서는 헤더와 푸터 정보를 삽입한다.
* find fit : 찾는 방법으로 first fit을 구현했다. first fit을 구현하기 위해서 explicit list의 시작 수조인 free\_listp부터 시작해서 헤더를 확인하여 get alloc이 0인 동안 while문을 반복하는데, 내부에서는 getsize로 크기를 구하고, 할당하려는 크기인 asize보다 크다면 해당 위치에 할당하면 되므로그때의 포인터를 반환한다. 그리고 만약 크기가 작다면, NEXT\_EX로 다음 free list로 넘어간다.
* coalesce : coalesce에서는 4가지 경우를 고려할 수 있는데, 앞뒤로 할당된 경우, 뒤만 할당된 경우, 앞만 할당된 경우, 앞뒤로 할당되지 않은 경우, 이때에 각각 맞춰 합춰준다. 그리고 이때 합춰주는 과정에서 푸터와 헤더의 정보를 수정하는 것 뿐 아니라 할당이 되어있지 않은 free block을 free list에서 제거하는 함수인 remove\_block을 호출해야한다. 그리고 새롭게 할당된 하나의 블럭을 다시 free list에 넣기 위해 insert\_block을 호출한다.
* insert\_block : 블럭을 삽입하는 알고리즘으로는 LIFO를 사용한다. 즉 가장 마지막에 삽입한애가 가장 앞이고, free\_listp와 연결되어야한다. 이를 위해서 해당 포인터의 PRE\_EX로는 null을 삽입하고 NEXT\_EX는 Free\_listp(현재 freelistp가 가리키는 블럭)을 할당하고 free\_listp의 PRE\_EX를 해당 블럭으로 연결한다. 그리고 free\_listp에 현재 블럭을 할당하여 가장 앞이 되도록 한다.
* remove\_block : 해당 블럭이 할당되었을 때, free에서 없애주는 함수다. 이때 경우를 두가지로 나누어야하는데, 첫 번째는 해당 블럭이 free list의 첫번째 블럭인 경우이다. 이는 해당 블럭의 PRE\_EX가 null인지 체크해서 확인한다. 이경우라면 free\_listp가 가리키는 부분을 다음인 NEXT\_EX로 바꾸고, NEXT\_EX의 PRE\_EX를 null로 만들어서 맨처음으로 만들어준다.

1. **구현 및 결과 및 성능 평가**결과적으로 event based, thread based 방식으로 주식 서버를 구현했다. event based의 기본적 흐름은 main함수에서select 함수를 이용하여 어떤 fd가 새로운input을 가지는지 확인하고, listenfd가 입력을 가진다면 connection을 accept해서 pool의 connfd 배열에 새로운 connfd를 추가하고, pending input이 있는 모든 connectfd의 요청을 처리하는 서버이다. multiclient 입장에서는 concurrent하지만, 실제 구현상으로는 sequential한 logical flow를 가지고 있다.thread-based 서버. sbuf라는 구조체를 이용하여 마스터 스레드와 피어 스레드가 connfd를 주고 받는데, 마스터 스레드가 클라이언트로부터의 연결을 accept하여 받은 connfd를 sbuf에 넣고 이를 available한 thread가 가져와서 요청을 처리하도록 했다. Event-based 서버와 달리, context-switch를 이용해 실제로 concurrent한 서비스를 제공한다.

* **2번의 구현 결과를 간략하게 작성**
* **미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가**

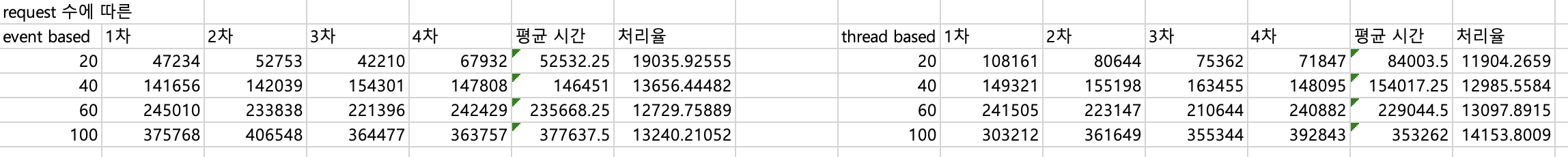
1. **성능 평가 결과 (Task 3)**측정 시점은 stockserver에서 구하며, 위의 구현에서 설명한 이벤트 기반에서의 pool.conn\_count, 스레드기반의 글로벌 변수 conn\_count가 0이 되는 시점을 기준으로 한다. 추가로 multiclient의 구현을 보면 주문을 넣은 후 usleep(1000000)으로 1초간 기다리는 함수가 존재하는데 이것이 정확한 성능 측정에 방해가 될 것을 고려하여 usleep(1000)으로 변경하여 실험했다.  **텍스트, 스크린샷, 폰트, 메뉴이(가) 표시된 사진

   자동 생성된 설명**실제 출력은 다음과 같다. 동시처리율은 위의 시간을 기준으로 구하는데 **동시처리율 : (client 수 \* client 당 order 수) / 소요시간(초단위)**위에 대입해보면 클라이언트 수는 10, 주문 수는 10, 소요시간은 31786/1000000이므로 대략 3146이 나온다. 이는 1초에 3146개의 요청에 대해서 다룰 수 있다는 말이다. 이 수치는 usleep 수치에 따라서도 달라질 수 있기에 절대적 수치보다는 상대적으로 수치를 비교하면서 분석할 필요가 있다. 또한 주문의 랜덤 요소, 커널의 스케줄링에 따라 시간이 조금씩 바뀌어서 각 측정 수치에 대해서 4번을 수행하고 그것의 평균 값을 가지고 동시 처리율을 분석했다.
2. **client 수 증가에 따른 동시 처리율  
   텍스트, 스크린샷, 라인, 그래프이(가) 표시된 사진

   자동 생성된 설명  
   **측정 시간과 그것을 처리율로 변환한 표, 그리고 그값들의 시각화이다.  
   앞서 예측과 다르게 thread 기반의 방식이 event based의 방식에 비해서 전반적으로 처리율이 높았다. 그런데 그 차이는 client 수가 증가함에 따라 더 커졌는데,미리 스레드를 만들어두고 그들을 통해 처리하는 concurrent 방식이 따른 thread 생성, 관리, syncrhonization 등의 오버헤드에도 불구하고 sequential한 event based 방식에 비해 높게 나타났다.

**request 수 증가에 따른 동시 처리율**

**텍스트, 스크린샷, 라인, 도표이(가) 표시된 사진

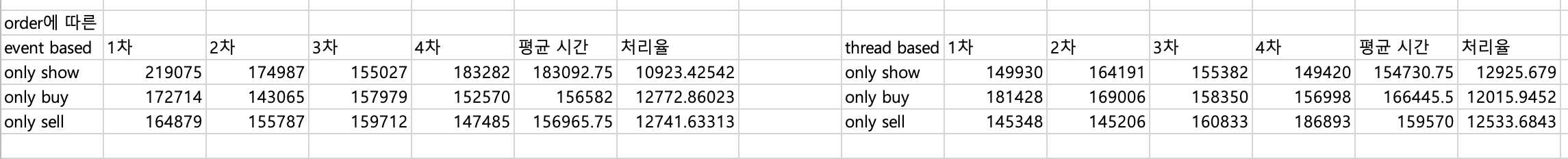
자동 생성된 설명  
**

위와 마찬가지로 그래프와 표다. 예상과 비슷했던 부분은 request 숫자가 증가할 수록 thread 기반 방식의 동시 처리율이 event based 방식보다 높아졌다는 것이다. 이는 역시 스레드를 미리 만들어두는 thread 기반의 방식이 가지는 이점이라고 생각한다. 한가지 예상과 달랐던 것은 request가 적을때는 event based의 방식의 처리율이 굉장히 높았는데 request가 증가할수록 처리율이 매운 낮아진다는 점이었다. 이는 아마도 event based의 방식이 결국 sequential한 방식이기 때문에 요청 수가많아질수록 처리 효율이 낮아지기 때문이라고 추측한다.

**client 요청 타입에 따른 동시 처리율**

텍스트, 스크린샷, 라인, 그래프이(가) 표시된 사진

자동 생성된 설명



예상한대로 read lock이 걸려있는 show만을 요청한 경우에는 thread based방식의 처리율이 event based보다 월등히 높았다. 세마포의 오버헤드가 있더라도 show 요청에 대해서 모두 동시 처리를 할 수 있기 때문이다. 반면 write lock이 걸려있는 only buy, only sell의 경우에는 모든 요청이 concurrent하게 처리될 수 없고 오히려 write lock과 다른 세마포 변수에 대한 오버헤드 때문에 event based 방식보다 낮은 처리율을 보였다.

**결론 :** event based와 thread based 접근을 정확하게 비교하기에는 multiclient의 usleep 그리고 os의 스케줄링 이슈등으로 인해 한계가 있다. 그럼에도 불구하고 위의 3가지 관점에서 비교했을 때, thread 방식은 다양한 오버헤드가 존재함에도 불구하고 실제로 concurrent하게 요청을 처리한다는 점에서 sequential 한 event based 방식보다 높은 효율 및 동시 처리율을 보였다.