**Circular Buffer 구현 및 실험**

**1. 서론**

FIFO 자료구조인 Circular Buffer를 C로 구현하고 std::queue와의 비교 실험을 진행하였다.

**2. 구현**

**2.1. cbuf.{c,h}**

**#define SUCCESS (0)**

**#define FAILURE (1)**

**#define NONE (-1)**

에러 처리와 예외 값 처리를 위한 상수다.

**typedef int Item;**

**#define NONE\_ITEM NONE**

원하는 데이터를 Item으로 설정하여 사용할 수있다. 이번 실험에서 Item은 int로 설정하였다.

**typedef struct CirBuf {**

**Item\* cbuf;**

**int buf\_size;**

**int head;**

**int tail;**

**} CirBuf;**

Circular Buffer를 정의하는 구조체다. 멤버 cbuf는 데이터를 저장할 배열이다. buf\_size는 cbuf가 가리키는 배열의 크기이다. head와 tail은 각각 버퍼의 시작과 끝을 가리키는 인덱스이다.

**int cbuf\_init(CirBuf\* cbuf, size\_t buf\_size);**

생성한 CirBuf 구조체의 포인터를 cbuf로 넣고, 버퍼 사이즈 buf\_size를 정해주면 cbuf의 멤버 cbuf에 메모리를 동적할당한다. 만일 메모리가 모자라 할당에 실패한다면 FAILURE를 반환한다. 그렇지 않으면 SUCCESS를 반환한다. 인자로 오는 cbuf는 후술하는 함수의 첫 인자 cbuf에 입력하여 사용한다.

**int cbuf\_deinit(CirBuf\* cbuf);**

생성한 CirBuf 구조체의 포인터를 cbuf로 넣으면 cbuf의 멤버 cubf의 메모리를 해제한다.

**int cbuf\_empty(CirBuf\* cbuf);**

cbuf가 비어있으면 0이 아닌 값, 그렇지 않으면 0을 반환한다. head와 tail이 같으면 빈 버퍼다.

**int cbuf\_full(CirBuf\* cbuf);**

cbuf가 꽉 차 있으면 0이 아닌 값, 그렇지 않으면 0을 반환한다. head가 tail바로 다음 칸을 카리키면 꽉 찬 버퍼다.

**int cbuf\_push(CirBuf\* cbuf, Item item);**

cbuf에 새로운 item을 삽입한다. 삽입에 성공하면 SUCCESS를 반환, 그렇지 않으면 FAILURE를 반환한다. 현재 tail이 가르키는 cbuf의 칸에 item을 복사하여 넣고, tail은 다음 칸을 가리키도록 1만큼 더한다. 만일 tail이 버퍼의 마지막 칸의 인덱스라면, item 삽입 후 tail은 버퍼의 첫번째 칸을 가리키게 한다. 버퍼가 꽉 찬 경우 실패한다.

**Item cbuf\_pop(CirBuf\* cbuf);**

cbuf에서 head가 가리키고 있는 item을 삭제하고 반환한다. 반환에 성공하면 item을 반환하고, 그렇지 않으면 NONE\_ITEM을 반환한다. 현재 head가 가르키는 item 반환하고, head는 다음 칸을 가리키도록 1만큼 더한다. 만일 head가 버퍼의 마지막 칸의 인덱스라면, item 삽입 후 tail은 버퍼의 첫번째 칸을 가리키게 한다. 버퍼가 빈 경우 실패한다.

**2.2. unit test: test.cpp**

cbuf 구현을 위해 다양한 경우를 테스트하였다.

**TEST(cbuf, push\_and\_pop)**

버퍼의 크기 안에서 아이템이 삭입되고 삭제되는 과정을 테스트하였다.

**TEST(cbuf, cannot\_pop\_from\_empty\_cbuf)**

빈 버퍼에서 아이템을 삭제할 경우, NONE\_ITEM을 반환하는 것을 테스트하였다.

**TEST(cbuf, cannot\_push\_to\_full\_cbuf)**

꽉 찬 버퍼에서 아이템을 삽입할 경우, FAILURE을 반환하는 것을 테스트하였다.

**TEST(cbuf, circular\_push)**

head나 tail이 버퍼의 끝에서 처음으로 돌아오는 circular push를 테스트하였다.

**3. 실험**

**3.1. 실험 구성**

std::queue와 성능을 비교하기 위해 두가지 실험을 설계했다. N개의 아이템(1~N)에 대해

1) 1 ~ N을 모두 삽입한 후에 모두 삭제하는 경우

2) 1 ~ N을 데이터 삭제와 삽입을 50% 확률로 무작위로 수행할 경우

cbuf와 std::queue에서 걸리는 시간을 측정하였다.

실험은 n번 반복 수행하였다. i번째 실험에서 N = (229 / n) \* i으로 설정하여 반복하여 실행에 걸린 시간을 측정하였다. main.cpp에서 실험을 하고,실험 결과를 yaml 포맷으로 stdout에 출력하였다. 출력된 결과를 파이썬 스크립트 plot.py를 이용하여 그래프로 나타내었다.

**4. 실험 결과**

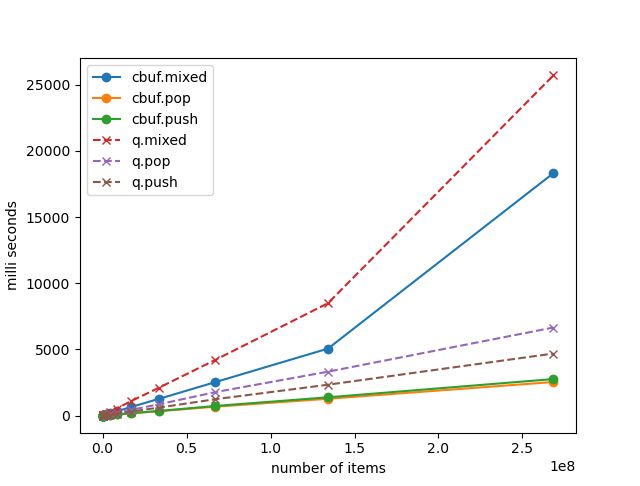


그림 1. 데이터에 대한 circular buffer, std::queue의 수행시간

cbuf가 붙는 결과는 circular buffer를 사용한 결과, q가 붙은 결과는 std::queue를 사용한 결과이다. pop, push는 3.1의 실험1)에서 각각 아이템을 삽입, 삭제하는데 걸린 시간이다. mixed는 3.1의 실험2) pop,push 혼합 작업을 수행하는데 걸린 시간이다.

표 1. 수행시간 상승 비율. 수행시간을 외삽하여 기울기를 측정하였다.

|  |  |  |
| --- | --- | --- |
| 데이터 \ 자료구조 | queue | cbuf |
| mixed |  |  |
| push |  |  |
| pop |  |  |

수행 시간은 std::queue와 circular buffer 모두 데이터수 N에 비례하는 선형으로 증가하며, 전체적으로 queue보다 circular buffer의 성능이 더 낫다. queue의 경우 연결리스트로 구현되기 때문에, circular buffer와 시간 복잡도는 동일하나 실제 수행 시간 성능에서 큰 차이를 보인다. 이에 비해 circular buffer는 연속적인 배열을 힙에 선언하는 방식이기 때문에 메모리 참조의 locality가 높아 캐시를 매우 효율적으로 활용할 수 있다. 이번에 구현한 circular buffer의 단점은 삽입하려는 데이터의 수가 버퍼 사이즈보다 클 경우 정보를 저장할 수 없다는 점이다.

**5. 결론**

이번 과제에서는 circular buffer를 구현하고, std::queue와의 성능을 비교해 보았다. 시간 복잡도는 queue와 circular buffer가 동일하나, 실제 수행 시간에 대해 연속된 배열을 이용하는 circular buffer가 더 나은 성능을 보였다.