Locking Wiki

IDEA

lock 구현을 위해서 process synchronization의 최신 알고리즘중 하나인 Lamport’s bakery algorithm을 사용할 것입니다. 해당 알고리즘의 pseudo code는 아래와 같습니다.

```
repeat
    choosing[i] := true;
    number[i] := max(number[0], number[1], ..., number[n - 1])+1;
    choosing[i] := false;
    for j := 0 to n - 1
        do begin
            while choosing[j] do no-op;
            while number[j] != 0
                and (number[j], j) < (number[i], i) do no-op;
        end;
        critical section

    number[i] := 0;

        remainder section
until false;
```

- **Description**
    - 해당 알고리즘에 대해서 간단히 설명하자면 이름이 bakery algorithm 답게 빵가게 들어가서 번호표를 뽑는 상황을 가정하는 알고리즘입니다. number는 현재 본인의 번호표로 마지막에 번호를 뽑은 사람이 가장 큰 수를 가집니다. 이는 atomic하게 보호되지 않기 때문에 같은 숫자가 여러번 나올 수 있습니다. 이를 for문의 두 번째 while문에서 뽑은 번호표와 본인의 thread id를 함께 비교하여 가장 처음 해당 스레드에 진입하고 또한 번호표 또한 가장 작은 단 하나의 thread만이 critical section에 진입을 하게 됩니다.

Implement

- 이는 위의 코드를 C언어의 코드로 변환한 것입니다. 이때 MAX_THREADS는 최대로 들어올 thread를 의미하므로 넉넉하게 설정하는 것이 권장됩니다.

```
#define MAX_THREADS 100
int number[MAX_THREADS]={0,};
int choosing[MAX_THREADS]={0,};

typedef struct{
    int tid;
}lock_args;

void lock(lock_args *arg){
    choosing[arg->tid]=1;
    __sync_synchronize();
    int max=0;
    for(int i=0;i<MAX_THREADS;i++){
        if(number[i]>max){
            max=number[i];
        }
    }
    __sync_synchronize();
    number[arg->tid]=1+max;
    __sync_synchronize();
    choosing[arg->tid]=0;
    for(int i=0;i<MAX_THREADS;i++){
```

```
        while(choosing[i]);
        __sync_synchronize();
        while((number[i]!=0) && ((number[i]<number[arg->tid]) || ((number[i]==number[arg->tid])
    }
}

void unlock(lock_args *arg){
    number[arg->tid]=0;
}
```

- **__sync_synchronize()**
  - 이는 GCC 컴파일러에서 제공하는 메모리 방벽으로 cpu에는 영향을 미치지 않지만 컴파일러가 load & store instruction의 순서를 바꾸는 것을 방지하는 코드입니다. 대량의 스레드가 생성이 되게 되면 compile가 효율성을 위해서 instruction의 순서를 바꿀 수 있게 되는데 이렇게 되었을때 위의 코드에서 choosing[arg→tid]=0; 이 코드 다음에 choosing이 가끔 1이 되는 상황을 발견하여 해당 코드를 추가하여 구현하였습니다.

Result

num_iters=1~999, num_thread=1~99의 임의의 숫자로 1만번 테스트를 진행했을때

shared_resource의 결과가 num_iters*num_thread의 값과 일치하는 것을 확인했습니다.

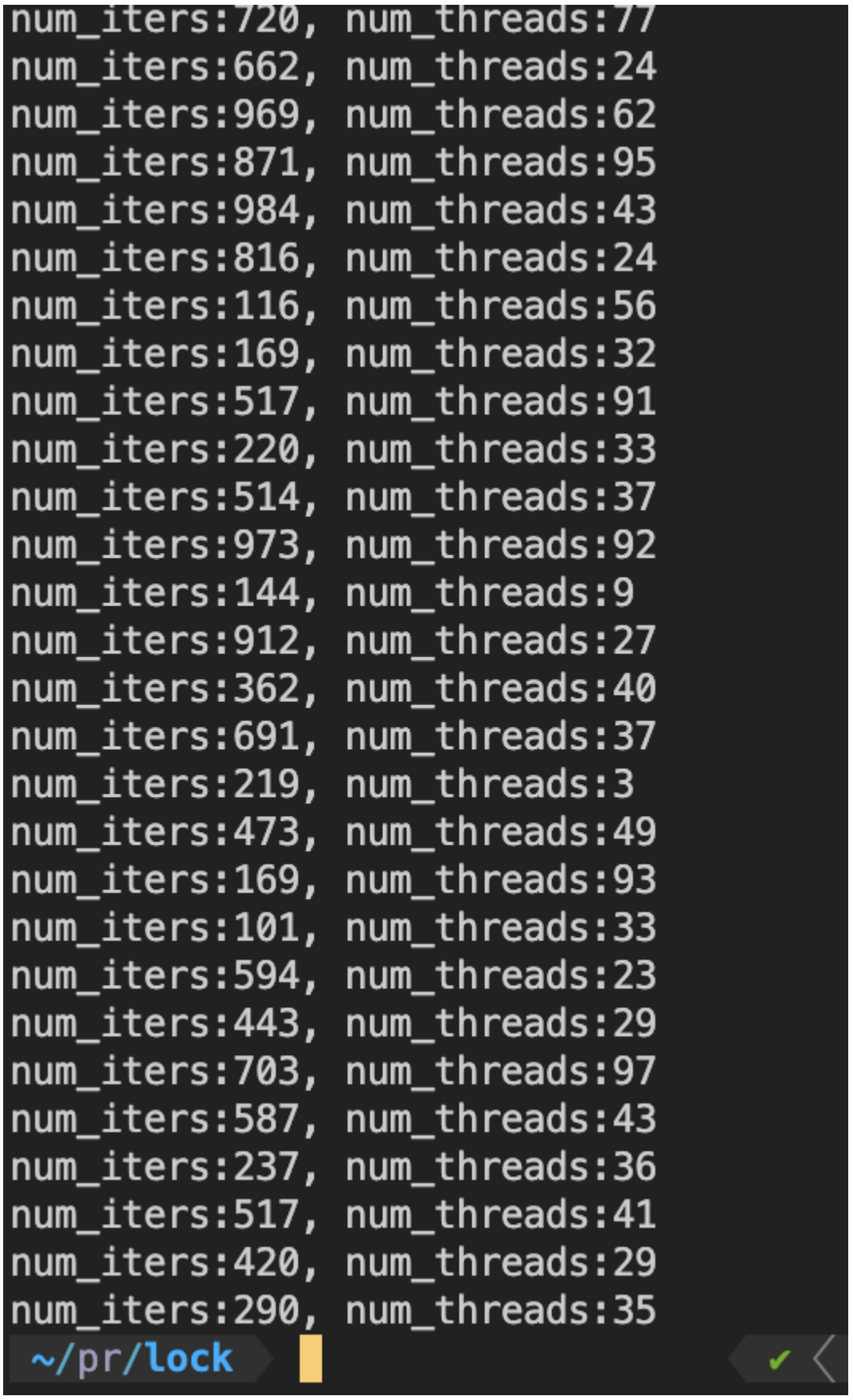
- **test code**

```
while(test_cnt<10000){
        test_cnt++;
        //init shared_resource
        shared_resource = 0;
        num_iters=rand()%1000;
        n=rand()%MAX_THREADS;
        pthread_t threads[n];
        int tids[n];

        printf("num_iters:%d, num_threads:%d\n", num_iters, n);
        lock_args* args=(lock_args*)malloc(sizeof(lock_args)*n);
        for (int i = 0; i < n; i++) {
            tids[i] = i;
            args[i].tid=i;
            pthread_create(&threads[i], NULL, thread_func, (void*)&args[i]);
        }

        for (int i = 0; i < n; i++) {
            pthread_join(threads[i], NULL);
        }
        for(int i=0;i<MAX_THREADS;i++){
            if(number[i]!=0 || choosing[i]){
                printf("(number[%d]:%d, choosing[%d]:%d) ", i, number[i], i, (int)choosing[i]);
            }
        }
        free(args);
        if(shared_resource!=n*num_iters){
            printf("shared: %d\n", shared_resource);
            break;
        }
    }
```

- **Result**



Trouble shooting/Thinking

- **다른 구현 아이디어(Peterson’s Algorithm for N Processes)**
  - 처음에는 다른 구현아이디어로 peterson algorithm을 N개의 프로세스에 대해서 진행하는 알고리즘을 생각했습니다. 이는 N개의 프로세스를 길이가 N인 queue에 집어넣고 큐의 마지막에 오는 process 들만 critical section에 진입하는 알고리즘으로 구현될 수 있었습니다. 이는 구현에는 성공했으나 단점이 있었습니다. 바로 bounded waiting이 보장이 되지 않는다는 것입니다.

```
i ← ProcessNo
for ℓ from 0 to N − 1 exclusive
    level[i] ← ℓ
    last_to_enter[ℓ] ← i
    while last_to_enter[ℓ] = i and there exists k ≠ i, such that level[k] ≥ ℓ
        wait
```

  - 큐에서 순서가 P0→P1→P2(끝)이라고 가정할때 P0가 P1, P2의 상태를 보면서 대기할때 P1, P2가 빠르게 critical section을 진입하고 다시 큐로 들어와서 P0보다 먼저 앞으로 나아가면 P1나 P2가 앞에 있어서 P0는 다시 대기를 하게되고 이동안에 또다시 P1, P2 둘중 하나의 아까와는 다른 프로세스가 앞에 존재하는 상황에 계속 반복적으로 발생할 수 있습니다.따라서 이는 bounded waiting을 보장하지 않습니다.