Clock Algorithm 구현 보고서

20191562 김성훈

요구사항 분석

LRU 에 근사한 hit rate 를 낼 수 있는 Clcok algorithm 을 구현한다.
이미 구현된 방식은 clock hand 라는 개념이 없고 무작위로 victim page 를 선발하는 알고리즘을 이용한다. 이러한 무작위 개념이 들어가면 seed 값에 따라 항상 결과가 다르며 또한 입력값에 따라 항상 결과가 다르다.
그래서 우리는 reference bit 라는 개념을 이용하고 clock hand 를 이용해서 순차적으로 모든 저장된 캐시를 확인할 것이다. 캐시를 담는 자료 구조는 Python 의 List 를 이용하고 modular 연산을 통해 저장 공간을 최대한 절약한다. 만약 Circular linked list 를 사용한다면 그만큼 포인터를 이용해서 불필요한 데이터가 많이 저장된다. 때문에 Python 의 List 내부 구현방식이 어떻게 되는지는 무시하고 순수 배열이라고 가정한다.
아래 설명은 구현에 대한 조건이다.

The OS checks if the currently-pointed to page P has a use bit of 1 or 0. If 1, this implies that page P was recently used and thus is not a good candidate for replacement. Thus, the use bit for P set to 0 (cleared), and the clock hand is incremented to the next page (P + 1). The algorithm continues until it finds a use bit that is set to 0, implying this page has not been recently used (or, in the worst case, that all pages have been and that we have now searched through the entire set of pages, clearing all the bits).”

위 조건 중 누락된 상황인 이미 해당 페이지가 존재하는 경우가 있는데 그러한 경우 이미 존재하는 페이지로 clock hand 를 바로 이동하고 reference bit 을 초기화 해줄 것이다.
reference bit 개념을 확장해서 LRU 에 더욱 근사한 결과를 낼 수 있는지 확인한다.
reference bit 의 최대 값을 1 이 아닌 옵션 값을 이용할 수 있도록 한 후 여러 케이스를 테스트 해볼 것이다.

요구사항 구현

clock hand 라는 개념이 기존에는 없기 때문에 해당 코드를 추가했다.

```
# track reference bits for clock
ref = {}
clock_hand = 0
```

hit 한 경우는 victim page 를 구하는 과정을 진행할 필요가 없기 때문에 찾은 위치로 clock hand 를 이동시킨다.

```
# hit 한 경우 clock hand 를 hit 한 페이지를 다음 페이지로 가르키게 한다.
elif policy == 'CLOCK':
    clock_hand = idx
pt:
```

miss 인 경우는 clock hand 를 순차적으로 이동시키면서 reference bit(use bit) 을 1 씩 감소시키는데 순차적으로 이동하는 과정에서 modualr 연산을 이용한다.

```
victim = -1
while victim == -1:
    # 현재 swap out 할 페이지가 찾아지지 않는다면 clock hand 를 돌리면서 use bit 를 하나씩 낮춘다.
    page = memory[clock_hand]
    if cdebug:
        print '  scan page:', page, ref[page]
    if ref[page] >= 1:
        ref[page] -= 1
    else:
        # this is our victim
        victim = page
        memory.remove(page)
        # victim 으로 선정된 곳을 제거한 후 그 위치에 page 를 insert 한다.
        memory.insert(clock_hand, n)
        break
    # 효율성을 위해 circular list based on array 를 이용한다. 그러므로 현재 저장된 count 를 초과하면 안된다.
    clock_hand = (clock_hand + 1) % count
```

compulsory miss 인 경우에는 굳이 clock hand 를 이동하며 victim page 를 찾는 과정을 진행할 필요 없기 때문에 바로 추가해준다.

```
# miss, but no replacement needed (cache not full)
victim = -1
count = count + 1

# CLOCK 알고리즘은 페이지가 꽉찬 경우는 replace 하기 때문에 꽉차지 않은 경우만 append 한다.
memory.append(n)
```

아래는 페이지가 캐시로 등록된 경우에 실행되는 코드들인데 주석의 내용처럼 항상 use bit 를 초기화하고 clock hand 도 현재 추가된 위치에서 다음 페이지로 넘어간다.

```
# CLOCK 알고리즘을 사용하면 use bit 는 구동시에 옵션으로 입력한 clockbits 로 설정한다.
# 또한 이미 존재하는 페이지이거나 새로 추가된 페이지인지는 상관이없다.
ref[n] = clockbits

# clockbits 로 use bit 를 설정한 뒤 clock hand 를 다음 페이지를 가르키도록 이동시킨다.
# 효율성을 위해 circular list based on array 를 이용한다. 그러므로 현재 저장된 count 를 초과하면 안된다.
clock_hand = (clock_hand + 1) % count
```

실행결과 분석

LRU 에 근접한 알고리즘 구현하는 것이 요구사항이기 때문에 LRU 의 최악, 대략적인 평균, 최상의 결과를 나타내는 간단한 테스트 케이스를 이용한다.

1. 최악 (worst.txt)
Cache size: 3
총 페이지 개수: 5 (1, 2, 3, 4, 5)
접근 순서: 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
hitrate: 0

2. 대략적인 평균 (average.txt)
Cache size: 3
총 페이지 개수: 5 (1, 2, 3, 4, 5)
접근 순서: 1, 1, 2, 3, 4, 5, 2, 3, 4, 5
hitrate: 20

3. 최상 (best.txt)
Cache size: 3
총 페이지 개수: 5 (1, 2, 3, 4, 5)
접근 순서: 1, 1, 2, 2, 3, 3, 4, 4, 5, 5
hitrate: 50

LRU 에 비해 대략적인 평균 테스트 케이스는 더 좋으며 최악과 최상은 동일하다. 과연 복잡한 결과는 LRU 와 비교했을 때 어떨지 살펴보겠다.

Cache size: 9
총 페이지 개수: 10 (0, 1, 2, 3, 4, 5, 6, 8, 9)
접근 순서: clockbits.txt
Clock 알고리즘은 clockbits 를 1 로 설정하는 경우 89.56 의 hitrate 를 나타내고 LRU 알고리즘은 89.75 의 hitrate 로 조금 더 좋다.

clockbits 에 변화를 준 경우
2: 89.89
3: 89.93
4: 89.69
5: 89.65
6: 89.69 (7 ~ 10000 까지 테스트한 결과 모두 89.69)

위 결과를 분석하면 페이지 접근 순서에 따라 clockbits 에 변화를 준 결과가 좋을 수도 나쁠 수도 있는 것을 알 수 있다. 또한 적절하게 큰 값을 설정한다면 hitrate 는 수렴하기 때문에 너무 높은 값도 좋지 않다. Cache size 와 page 개수에 따라 적절히 설정해야 된다.