페이지 폴트를 이용한 Userspace Read-Copy Update 기법 설계 및 구현

김인혁, 신은환, 엄영익
성균관대학교 정보통신공학부
e-mail:{kkojiband,comsky,yieom}@ece.skku.ac.kr

Design and Implementation of Userspace Read-Copy Update scheme using Page Faults

Inhyuk Kim, Eunhwan Shin, Youngik Eom
School of Information and Communication Eng., Sungkyunkwan University

요 약

멀티코어의 등장과 병렬 프로그래밍의 확산으로 lock-free 동기화 기법에 대한 관심과 필요성이 더욱 커지고 있지만, 대부분의 lock-free 동기화 기법들이 구현 복잡도와 동작시 오버헤드로 인해 실제 활용되는 사례는 미비하다. 하지만, RCU(Read-Copy Update) 기법의 등장으로 다양한 운영체제에서 이를 구현하여 활용하고, 최근에는 게임 서버와 같은 응용 프로그램에서도 이를 활용하려는 시도가 늘어나고 있지만, 기존에 제안된 URCU(Userspace RCU) 기법들은 메모리 순서오류 문제 해결을 위한 메모리 장벽 호출 및 reader와 updater 간의 IPC 등으로 충분한 성능을 보여주지 못하고 있다. 이에 본 논문에서는 페이지 폴트를 이용한 URCU 기법을 제안하고, 이를 구현하여 기존의 URCU 기법들과 실험을 통하여 평가하였다.

1. 서론

최근 프로세서의 발전 방향은 코어 속도 증가가 한계에 다다름에 따라 여러 개의 코어를 병렬로 연결하는 방식으로 바뀌고 있다. 서버 및 데스크톱에서 주로 사용되는 인텔의 x86 계열 프로세서들뿐 아니라 임베디드 환경에서 주로 사용되는 ARM 계열 프로세서들도 멀티코어 형태의 프로세서들을 출시하여 활용 범위를 넓혀가고 있다. 하지만 멀티코어의 성능을 제대로 활용하기 위해서는 기존의 소프트웨어를 병렬 처리에 알맞은 구조로 변경해야하는 어려움이 있다.

전통적인 순차 프로그램을 병렬 프로그램으로 변경하는 것은 힘든 작업이다. 개발자에 의해 병렬 처리가 필요한 부분을 스레드로 나누고 공유 데이터에 대해 빠짐없이 동기화를 해주는 등 수많은 노력과 검증이 필요하게 된다. 특히 동기화는 오버헤드뿐 아니라 데드락, 라이브락, 우선순위 역전 현상 등 프로그램의 부피가 커질수록 해결이 어려운 수많은 문제들을 야기시킨다. 이에 뮤텍스 기반의 동기화 기법 외에 lock-free 동기화 기법들에 대한 연구가 꾸준히 진행되어 왔다[1].

대표적인 lock-free 동기화 기법에는 transactional memory와 read-copy update가 있다. Transactional memory는 임계 영역 내에서의 메모리 접근이 다른 스레드에 의해 방해를 받을 경우 임계 영역 내에서의 메모리 접근을 모두 취소하고, 임계 영역 진입 전으로 복구하는 방식이다. 하드웨어 및 소프트웨어 기반의 다양한 transactional memory 기법들이 있지만 오버헤드가 크기 때문에 실용화되지는 못 하고 있다. 그에 반해 read-copy update는 읽기 작업이 상대적으로 많을 경우, 읽는 쪽의 오버헤드를 최소화하여 전체적인 성능을 높이는 기법이다. 이는 리눅스 커널, K42 등 다양한 운영체제에 적용되어 그 효과를 인정받고 있으며 점점 활용 범위를 넓혀가고 있다[2,3].

최근에는 게임 서버 등 다양한 사용자 프로그램에 read-copy update 기법을 적용하려는 시도가 있지만, 커널 레벨의 read-copy update와 달리 유저 레벨에서는 컴파일러 순서오류, 메모리 순서오류 문제 해결을 위한 메모리 장벽 호출 및 reader와 updater 간의 IPC 등으로 효과적인 구현 방식이 제시되지 못한 상태이다. 이에 본 논문에서는 새로운 형태의 유저 레벨 read-copy update 구현 기법을 제시하고, 이를 기존의 기법들과 비교하여 그 우수성을 확인하였다[4].

* 본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음. (NIPA-2010-(C1090-1021-0008))

본 논문의 구성은 다음과 같다. 2장에서는 다양한 lock-free 동기화 기법들에 대해 소개하고, 3장에서는 커널 레벨 및 유저 레벨의 read-copy update 구현 기법들을 비교/분석한다. 4장에서는 본 논문에서의 제안 기법을 소개하고, 5장에서는 실험을 통해 기존의 기법들과 비교하였다. 마지막으로 6장에서는 결론 및 향후 연구 계획에 대해 소개하였다.

2. Lock-free 동기화 기법

2.1. Transactional memory

Transactional memory는 데이터베이스의 트랜잭션 개념을 메모리에 적용한 기법으로, 트랜잭션이 시작된 후의 메모리 접근을 임시로 기록해두었다가 트랜잭션이 완료된 후 한 번에 적용하는 방식이다. 마지막 적용하기 전에는 트랜잭션이 시작된 후에 접근한 메모리에 다른 누군가의 접근이 있었는지 확인하고, 만약 다른 누군가의 접근이 있었다면 적용은 실패하고, 트랜잭션 시작 후에 임시로 기록된 내용들은 버려지게 된다.

Transactional memory를 구현하는 방식은 크게 하드웨어에 기반한 방식과 소프트웨어에 기반한 방식으로 나뉜다. 하드웨어에 기반한 방식은 캐시를 이용하여 메모리로의 적용을 미루는 형태로 이루어진다. 이러한 방식은 캐시 용량 문제, 문맥전환시 처리 문제 등 다양한 제약사항으로 실제 적용되기에는 무리가 있다. 소프트웨어에 기반한 방식 또한 다양한 기법으로 메모리 접근 내용을 임시로 저장하였다가 나중에 한 번에 적용하는 형태로 제안되었지만 구현 복잡성, 제한된 적용 범위, 오버헤드 등 실제 적용되기에는 무리가 있다[2,5].

2.2. Read-Copy Update

RCU(Read-Copy Update)는 공유 데이터를 읽는 스레드의 오버헤드를 최소화하는 방식이다. 공유 데이터를 갱신하는 스레드는 기존의 데이터를 읽는 스레드가 없는걸 확인할 때까지 데이터를 해제하는 것을 미루게 된다. 이 과정은 다음 그림 1과 같다.

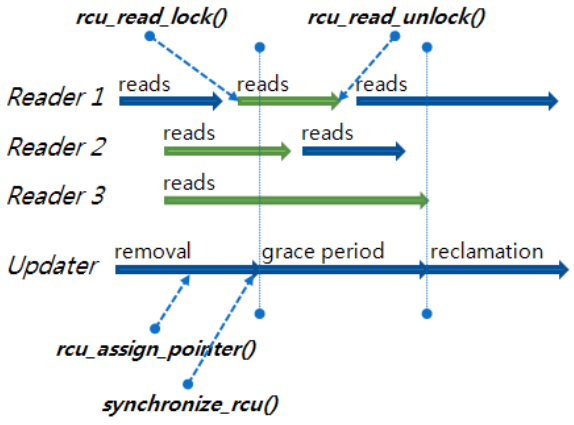

그림 1 RCU 기본 개념

위의 그림과 같이, reader는 어떠한 락도 없이 공유 데이터를 접근하고, updater가 공유 데이터를 변경할 때는 등록된 모든 reader들이 QS(Quiescent State)를 지나 기존의 데이터에 대한 접근을 완료했음을 확인 후 기존의 데이터를 해제한다[6,7].

다음은 read-copy update를 이용한 reader와 updater의 기본 구조이다.

```
1  void reader() {
2     rcu_read_lock();
3     x = rcu_dereference(g);
4     ...
5     rcu_read_unlock();
6  }
7
8  void updater() {
9     new = malloc(...);
10    new->contents = new value;
11    old = g;
12    rcu_assign_pointer(g, new);
13    synchronize_rcu();
14    free(old);
15 }
```

위에서 보인바와 같이, reader는 rcu_read_lock()/unlock() 함수를 이용하여 공유 데이터를 읽는 부분의 시작과 끝을 알리고 rcu_dereference() 함수를 이용하여 현재 공유 데이터에 대한 포인터를 얻어온다. Updater는 새로운 공유 데이터를 초기화 한 후, rcu_assign_pointer() 함수를 이용하여 공유 데이터를 교체하고, synchronize_rcu() 함수를 이용하여 모든 reader들이 QS를 지나는 것을 확인한 다음 기존의 데이터를 해제한다. 다음 장에서는 주요 함수들을 어떤 형태로 구현하는지를 알아보도록 하겠다.

3. Read-Copy Update

3.1. Kernelspace RCU

리눅스 커널이 사용하는 RCU는 rcu_read_lock()/unlock()에서 단순히 preempt_disable()/enable()을 호출한다. 즉, 임의의 프로세서에서 스케줄링이 발생하게 되면 해당 프로세서에서는 QS를 지났다고 판단할 수 있고, 읽기 작업을 수행하는 스레드의 수와 상관없이 시스템에 존재하는 프로세서의 수만큼 QS가 호출되면 grace period는 끝나게 된다. 이러한 방식은 구현이 간단한 장점이 있지만, 모든 프로세서에서 한 번씩 스케줄링이 발생할 때까지 grace period가 지속된다는 단점이 있다[6,8].

3.2. Userspace RCU

URCU(Userspace RCU)는 다양한 방식이 구현되어 있다. 가장 단순한 형태는 reader와 updater가 공유 변수를 이용하여 QS를 확인하는 방법이다. 이 방법은 컴파일러 순서오류, 메모리 순서오류 문제로 인해 reader가 매번 메모리 장벽을 이용하여 공유 변수를 확인해야하는 단점이 있다. 하지만 이를 해결하기 위해 updater가 새로운 공유 데이터를 등록할 때 모든 reader들에게 시그널을 보내고 시그널을 받은 reader들이 메모리 장벽을 이용하여 QS를 호출하는 방법이 제안되었고, 시그널을 보내는 비용을 줄이기 위해 커널에 새로운 시스템콜을 추가하여 현재 동작중인 모든 프로세서에게 IPI를 이용하여 메모리 장벽을 호출하게 하는 방식이 제안되었다. 하지만 이들은 여전히 reader 측의 오버헤드가 크다는 단점과 updater에서 시그널 혹은 IPI를 이용하여 모든 reader에게 메모리 장벽을 호출하도록 알려야 한다는 단점이 있다[9,10,11].

4. 페이지 폴트를 이용한 Userspace RCU

본 논문에서 제안하는 RCU는 QS를 확인하는 방법으로 페이지 폴트를 이용하였다. 이를 위해 공유 데이터는 특별히 관리되는 페이지에 위치하고, 기본 개념은 다음 그림 2와 같다.

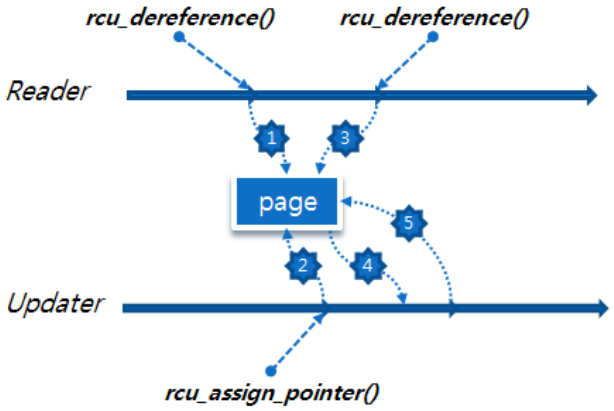

그림 2 페이지 폴트를 이용한 URCU 기본 개념

위의 그림과 같이, reader는 grace period 구간이 아닐 때는 다른 URCU와 달리 아무런 추가 작업도 하지 않는다(1). 만약 updater가 공유 데이터를 변경할 때는 해당 페이지를 가상주소공간에서 제거하고 모든 reader들이 QS를 지나기를 기다리게 된다(2). Reader가 grace period 구간에서 공유 데이터를 읽게 되면 페이지 폴트가 발생하게 되고(3), QS를 지났다는 사실을 updater에게 알린다(4). 그리고 모든 reader들이 QS를 지나게 되면(grace period 구간이 끝나면) updater는 해당 페이지를 다시 가상주소공간에 연결시킨다(5). 제안 기법의 장점은 reader가 QS를 확인하기 위해 메모리 장벽을 전혀 사용할 필요가 없고, updater도 모든 reader에게 IPC를 통해 공유 데이터 변경을 알릴 필요가 없다는 것이다.

Updater가 가상주소공간에서 제거한 페이지를 reader가 접근할 때는 grace period 구간이 끝날 때까지 해당 페이지를 가상주소공간에 연결하지 못하므로 다음과 같은 과정을 통해 해결한다. Reader가 공유 데이터를 읽는 부분과 grace period 구간에서 발생한 페이지 폴트를 처리하는 주요 코드는 다음과 같다.

```
1  static void *rcu_dereference(void *g) {
2    void *t;
3    asm volatile ("movl (%%edx), %%eax₩n"
4          : "=a" (t) : "d" (g));
5    return t;
6  }
7
8  void do_page_fault(struct pt_regs *regs, ...) {
9    if (address in grace period) {
10     regs->ip += 2;
11     regs->ax = new value;
12     if (end of grace period)
13       wake_up(updater thread);
14   }
15 }
```

Grace period 구간이 아닐 경우는 reader는 정상적으로 공유 데이터를 읽어들인다. 하지만 grace period 구간일 경우는 reader가 공유 데이터를 읽게 되면 페이지 폴트가 발생하게 되고, 페이지 폴트 핸들러에서 reader는 updater에게 새로운 공유 데이터에 접근했음을 알리게 된다(13). 그리고 해당 명령어를 완료하기 위해 IP를 강제로 증가시키고(10), EAX 레지스터에 공유 데이터를 넘겨주게 된다(11). 이렇게 함으로써 해당 페이지를 가상주소공간에 다시 연결시키지 않고도 공유 데이터에 대한 접근을 완료할 수 있게 된다.

5. 구현 및 실험 평가

제안 기법은 리눅스 환경에서 구현되었고, URCU에서 제공되는 기법들과 비교/평가하였다. 실험 환경은 다음과 같다.

CPU	Intel i7 QuadCore
Memory	4 Gbyte
Smart Cache	8192 Kbyte
OS	Linux kernel 2.6.33
Compiler	GCC 4.3.3
Library	GNU C library 2.9
URCU	Userspace RCU 0.4.6

위의 실험 환경에서 제안 기법을 pthread에서 제공하는 mutex/rwlock과 URCU-QSBR, URCU-SIGNAL과 비교한 결과는 다음 그림 3과 같다. URCU-QSBR은 이상적인 성능을 판단하기 위한 기법으로 reader는 메모리 장벽을 전혀 사용하지 않고, 주기적으로 QS를 호출하는 방식으로 구현되어 있다.

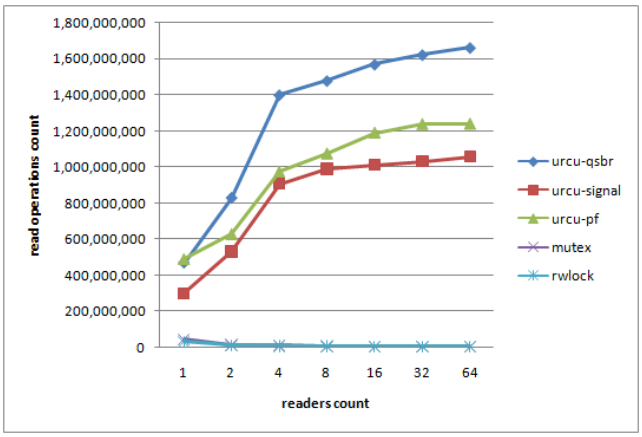

그림 3 URCU 성능 평가

위의 실험 결과와 같이, reader가 1개일 경우에는 제안 기법(URCU-PF)이 URCU-QSBR 보다도 좋은 성능을 보이는 것을 알 수 있다. 이는 이상적인 성능을 보이는 URCU-QSBR의 경우 reader 측에서 주기적으로 QS를 호출하는 등의 작업이 있지만, 제안 기법의 경우에는 reader 측의 추가 작업이 전혀 없기 때문이다. Reader가 2개 이상일 경우에는 제안 기법이 다른 URCU 기법보다는 좋은 성능을 보이지만, URCU-QSBR 보다는 안 좋은 성능을 보이는데, 이는 grace period 구간 동안 모든 reader들이 공유 데이터를 읽을 때마다 페이지 폴트가 발생하기 때문이다.

6. 결론

본 논문에서는 대표적인 lock-free 동기화 기법인 RCU를 사용자 계층에서도 효과적으로 사용할 수 있는 기법을 제안하였다. 제안한 페이지 폴트를 이용한 URCU 기법은 실제 적용 가능한 다른 URCU 기법들보다 우수한 성능을 보이고 있다. 향후에는 grace period 구간 동안 reader들이 공유 데이터를 읽을 때마다 페이지 폴트가 발생하는 문제를 해결하면 이상적인 성능을 보이는 URCU-QSBR 보다도 좋은 성능을 보일 것으로 기대된다.

참고문헌

[1] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in Linux," Proc. of the Linux Symposium, 2002.

[2] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, "Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system," Proc. of the 3rd Symposium on Operating System Design and Implementation, pp. 87-100, 1999.

[3] K. Fraser, "Practical Lock-Freedom," Ph.D. dissertation. King's College, University of Cambridge, 2003.

[4] P. McKenney, D. Sarma, I. Molnar, and S. Bhattacharya, "Extending RCU for Realtime and Embedded Workloads," Proc. of the Linux Symposium, pp. 123-138, 2006.

[5] C. Cascaval, C. Blundell, M. Michael, H. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software Transactional Memory: Why is it only a research toy?," ACM Queue, 2008.

[6] P. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell, "Read-Copy Update," Proc. of the Linux Symposium, 2001.

[7] P. McKenney and J. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," Proc. of Parallel and Distributed Computing and Systems, pp. 509-518, 1998.

[8] P. McKenney, "Exploiting deferred destruction: An analysis of read-copy-update techniques in operating system kernels," Ph.D. dissertation, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[9] M. Desnoyers, P. McKenney, A. Stern, M. Dagenais, and J. Walpole, "User-Level Implementations of Read-Copy Update," IEEE Transactions on Parallel and Distributed Systems, 2009.

[10] T. Hart, P. McKenney, and A. Brown, "Making lockless synchronization fast: Performance implications of memory reclamation," Proc. of the 20th International Parallel and Distributed Processing Symposium, 2006.

[11] T. Hart, P. McKenney, A. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," Journal of Parallel and Distributed Computing, vol. 67-12, pp. 1270-1285, 2007.