제19회 한국정보처리학회 춘계학술발표대회 논문집 제10권 제1호 (2003. 5)

하이브리드 멀티 프로세스 멀티 스래드 방식 웹서버의 시스템 호출 오버해드 분석

염미령
수원여대 컴퓨터 응용학부 디지털 콘텐츠과
e-mail:miryeom@suwon-c.ac.kr

An Analysis of System calls for Web Server : Apache 2.0 MPM-worker

Mi-Ryeong Yeom
Dept of Digital Contents, Suwon Women's College

요 약

웹 서버는 CPU time의 대부분인 75~78%를 시스템 코드에서 소비하며 사용자 코드에서는 생각보다 많은 시간을 소비하지 않는다. 이것은 웹 서버의 성능에 운영체제가 많은 영향을 끼치고 있음을 암시하는 것이다. 본 논문에서는 Linux Trace Toolkit[7]를 이용하여, 하이브리드 멀티 프로세스 멀티 스래드 방식의 아파치 웹 서버가 구동 중인 동안 호출되는 시스템 호출의 동작 과정과 역할에 대해 알아보고 어떤 시스템 코드에서 오버헤드가 큰지를 분석하였다.

1. 서론

인터넷에 연결된 웹 사이트의 정보를 수집하는 "The Netcraft Web Server Survey" 보고서에 의하면 아파치 웹서버는 2003년 2월 현재 구동중인 인터넷의 웹 서버 중 약 66.75%를 점유하고 있으며 사용량이 계속 증가 추세에 있다.

아파치 웹서버는 다양하고, 강력한 기능을 갖추었으며 매우 안정적이며 무료 소프트웨어라는 점에서 점점 대중성을 넓혀가고 있다. 아파치는 소스 코드가 공개된 오픈 프로젝트(Open project)방식으로 진행 중이며, 사용자가 성능을 더 좋게 하기 위하여 자유로이 코드를 수정할 수 있다. 현재까지는 안정버전 1.3.x까지 개발되었다. 동시에 이와는 별도로 새로운 기능을 보강하고 성능 향상을 위해 많은 부분에 수정을 가한 2.0버전의 개발이 진행중이다. 2.0에서는 웹서버의 성능을 향상시킬 수있는 시스템 호출들이 대부분 채택되었지만, 아이러니컬하게도 신버전 2.0은 구버전 1.3의 성능을 앞서지 못한다. 그 밖에 웹서버의 성능에 영향을 미치는 요소들은 매우 복잡하게 얽혀있지만 본 논문에서는 하이브리드 멀티프로세스 멀티 스래드 방식을 채택하는 2.0 버전이 1.3에 비해 달라진 시스템 호출들을 알아보고 이들이 갖는 오버해드에 대해 분석해 보았다. 본 논문에서는 웹 서버가 구동 중인 동안 어떤 시스템 코드에서 많은 오버해드를 갖는지 분석하기 위해 위해 Linux Trace Toolkit[10]를 이용하였다.

본 논문은 제 1절 서론, 제 2절 관련연구, 제 3절 아파치 2.0의 구동 방식, 제 4절 시스템 호출 분석 그리고 마지막으로 제5절 결론으로 구성된다.

2. 관련연구

웹 서버의 성능 향상을 위해 많은 다양한 연구들이 이루어지고 있다. 먼저 사용자 레벨에서 웹 서버를 최적화 시키려는 연구들을 살펴보면, Hu et al.[1]는 아파치 초기 버전의 분석을 통해 다양한 최적화를 구현함으로써 성능을 향상시켰는데 아파치

1.3과 2.0에는 이러한 연구의 결과가 많이 반영되었다. Nahum et al.[2]에서는 기존의 다양한 최적화 socket API와 시스템 호출이 웹 서버의 성능을 얼마나 향상시킬 수 있는지를 비교 평가했다. 또한 커널 레벨에서도 웹 서버를 최적화시키려는 연구가 이루어 졌는데, Kaashoek et al.[4][5]는 디스크 버퍼 캐쉬와 네트워크 전송 버퍼를 통합시키고 인터넷 체크섬 계산의 오버헤드를 줄이기 위해 HTTP 문서의 체크섬을 미리 계산하는 등 HTTP 전용 운영체제를 제시하였다. Druchel et al.[3]에서는 웹 서버는 기존의 OS에 잘 적응하지 못한다는 단점을 해결하기 위해 zero-copy 데이터 와 체크섬 캐싱을 제공하는 커널 OS를 제시하였다. 최근 들어 실험 및 상용 웹 서버들은 TCP/IP 스택에 HTTP 프로토콜이 통합되는 추세이며 커널 모드로 동작되므로 사용자 모드의 웹 서버에 비해 성능이 매우 우수하다[6].

3. 아파치 2.0의 구동 방식

아파치 2.0은 기본적으로 하이브리드 멀티 스래드 멀티 프로세스 방식(MPM-wroker)으로 구동된다. 2.0은 1.3에서 지원되는 프리포킹 방식 이외에 다른 멀티 프로세싱 모듈(MPM:MultiProcessing Module)들을 2가지 더 지원하므로써 시스템 환경에 적절한 모델을 채택하도록 한다. 본 절에서는 2.0에서 제공되는 실행 모듈에 대해 설명할 것이다.

3.1 MPM-Pre Forking

프리포킹 모듈은 아파치 1.3의 디폴트 구동 방식이며 2.0에서도 1.3과 거의 같은 방식으로 동작한다. 주프로세스에 의해 생성된 여러 개의 프로세스가 각각 클라이언트의 요청에 따라 서비스를 제공하는데 클라이언트의 요청(Request)을 대기(listen)하고 있다가 요청허락(accept operation)을 수행함으로써 클라이언트의 요청을 받아들인 후 필요한 서비스를 한다. 이러한 일련의 작업은 클라이언트의 요청 수에 비례한다. 그러므로, 실행하는 프로세스의 개수와 각 프로세스가 처리할 수 있는 요청의 수 등을 설정 파일을 이용하여 정할 수 있다. 프리포킹 웹서버는 성능 향상을 위해서 최초 프로그램 실행 시부터 기본적으로 단일 프로세스가 아닌 일정 수의 프로세스가 실행되게 되며(StartServers), 또한 여분의(idle) 서버 프로세스들을 항상 풀로 유지하고자 함으로써 클라이언트의 요청이 집중될 경우에도 서비스에 차질이 없도록 하였는바, 이는 요청들의 집중 시에도 서비스할 새로운 프로세스가 생성되는 시간 동안 기다리지 않게 함으로써 실행의 효율성을 제고시킨다. 프로세스의 생성은 StartServers의 숫자 만큼 초기화되어 시작된 후 로드가 높아지면 빠른 비율로 children을 생성하는데 MinSpareServers 셋팅을 만족할 때까지 진행 된다. 전체 프로세스의 수는 MaxServers를 넘지 못한다.

아파치 1.2 버전까지는 1초에 하나의 프로세스가 생성되는 one-per-second rule이 적용 되었으나 아파치 1.3 버젼 부터는 조금 완화되어 Child를 1개 생성한 후, 1초를 기다리고 2개를 생성하고 또 1초를 기다리고 다음으로 4개를 생성하는 방식으로 수행된다. 생성된 프로세스는 프로세스풀로 관리되고 프로세스 스케쥴링은 운영체제에 의존하며 애플리케이션 레벨에서는 특별히 제어하지 않는다.

3.2 MPM-worker

이 모듈은 대부분의 유닉스 계열에서 기본 모듈로 셋팅방식으로 하이브리드 멀티 스래드 멀티 프로세스 웹 서버를 구현한다. 주프로세스의 역할은 자식 프로세스들을 생성하는 것이며, 생성된 각 자식 프로세스들은 정해진 갯수의 스레드를 생성시킨다. 이 스래드들은 컨넥션을 리슨하고 있다가 클라이언트의 요청이 도착하면 서비스한다. 아파치는 항상 모든 프로세스내의 아이들(idle)한 스래드의 개수를 주시하고 있다가 프로세스들을 죽이거나 새로이 포크(fork)한다. 즉 서버의 로드 상태에 따라 새로운 프로세스를 새로이 생성시킬 수 있는 모델이다.

3.3 MPM-perchild

이 모듈은 MPM-worker과 마찬가지로 하이브리드 멀티 스래드 멀티 프로세스 웹 서버를 구현한다. 주프로세스의 역할은자식 프로세스들을 생성하는 것이며, 생성된 각 자식 프로세스들은 스레드들을 생성시킨다. 이 모듈은 실행될 프로세스의 개수가 고정되어 있어서 실행 중에 새로운 프로세스의 생성은 일어나지 않는다. 다시 말하면, 자식 프로세스들은 웹 서버가 처음 구동될 때 모두 생성되므로 더 이상 증가되지 않는다. 그러나, 스래드는 로드 상태에 따라 각 프로세스내에서 생성될 수 있는 최대 수까지 새로이 생성될 수 있다.

4. MPM-worker의 시스템 호출 분석

아파치 1.3에서는 CPU의 22%~25%정도를 사용

자 코드에서 소비하며 그 외의 대부분의 시간을 시스템 코드에서 소비한다(그림 1). 그 이전 버전의 아파치에서도 비슷한 모습이었다. 그러나 MPM-worker (그림 2)에서는 1.3에 비해 좀더 변화의 폭이 넓다(23~36%). 아파치 1.3에서는 정적 요청이 많을수록 시스템 코드의 수행 시간이 많아지는 반면 MPM-worker는 동적 문서의 요청이 많을수록 시스템 코드의 수행 시간이 많아지는 모습을 보인다.

본 논문에서는 MPM-worker가 구동 중인 동안에 어떤 시스템 코드에서 많은 오버헤드를 갖는지 분석하기 위해 위해 Linux Trace Toolkit[7]을 이용하였다. LTT는 최소한의 메모리와 LTT time 오버헤드(<2.5%)으로 시스템에서 발생되는 각 이벤트의 정보를 수집한다. 본 논문에서는 LLT가 출력하는 커널 트레이스를 통해 시스템 호출의 시간 비율을 비교하였다

다음의 그림 3은 아파치 2.0이 MPM-worker로 동작 될 때 측정된 시스템 호출의 시간 비율 분석 그래프이다. 정적 문서의 비율을 0%에서부터 100%일 때까지 증가시키며 측정하였다. 그래프에는 각 실험의 측정에서 상위 몇 개의 시스템 호출에 대해서만 표현하였으며 하위 1%에 속하는 시스템 호출은 etc로 표현하였다.

정적 문서의 비중이 많을수록 select()의 호출 비용은 상당히 커진다. MPM-worker에서는 소켓 또는 파일에 쓰기 동작이 실패했을 경우 일정량의 시간을 기다린 후 재전송 하는데, 이때 select() 시스템 호출을 통해 기다릴 시간만큼의 타임 아웃을 셋팅한다. 본 논문의 실험 데이터를 살펴보면, 정적 문서 100%일 때 나가는 패킷의 수는 CGI 100%일 때에 비해 약 4.6배 이상 많다. 그러므로, 나가는 패킷의 수가 많은 정적 문서 100%를 수행할 때 소켓에 쓰기 작업을 실패할 확률이 더 높다.

CGI 요청의 비중이 커질 수록 read() 시스템 호출 오버헤드는 전체 커널 시스템 호출의 과반수 이상을 차지한다. 왜냐하면, 아파치는 파이프를 통해 CGI 프로그램과 통신을 하기 때문이다. CGI 프로그램이 자신의 프로그램 수행 결과를 파이프에 쓰면, child 스래드는 파이프에서 read() 시스템 호출을 통해 읽어온다. 그러므로, 동적 프로그램의 수행이 많게 될수록 파이프 읽기 동작이 잦아진다. Read() 시스템 호출의 오버헤드를 줄이기 위해서는 파이프 읽기를 최적화시키는 작업이 요구된다. MPM-worker 모듈에서 스래드에만 관련되는 시스템 콜은 modify_ldt, sched_setscheduler(), rt_sigsuspend()이다. 위의 시스템 호출 중 rt_sigsuspend()를 제외하면 나머지 시스템 호출의 오버헤드는 아주 작았다. 각 스래드들은 활성될 때 modify_ldt()를 호출함으로써 자신들이 접근하게 될 메모리 공간에 대한 설정을 알아낸다. 그 다음으로 sched_setscheduler() 시스템 호출을 통해 프로세스의 스케쥴링 방식과 동일한 SCHED_OTHER 방식으로 셋팅된다. (리눅스에서는 CPU 스케쥴링 방식으로 time-sharing 정책인 SCHED_OTHER을 디폴트로 사용한다.)

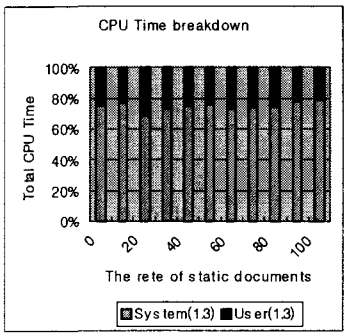

그림 1 아파치 1.3에서의 CPU Time

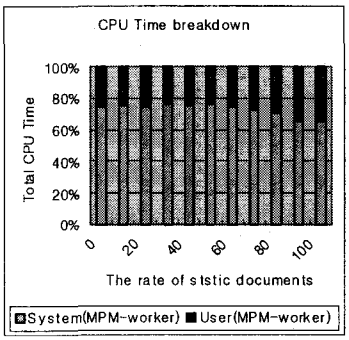

그림 2 MPM-worker에서의 CPU Time

rt_sigsuspend() 시스템 호출은 스래드들 간에 synchronization을 수행하기 위해 필요하기 때문에 모든 비율의 실험에서 거의 일정량을 차지한다. 각 스래드들은 같은 자원을 얻기 위해 경쟁하는데 자원을 획득하지 못한 스래드들은 rt_sigsuspend()를 호출하므로써 정지된다(suspend). 이렇게 정지된 스래드들은 자원을 사용할 수 있을 때까지 기다리게 된다. rt_sigsuspended()를 호출 하는 가장 대표적인 곳은 다음과 같다. 아이들(Idle)한 스래드들이 소켓에서 기다리고 있는 컨넥션이 있는지를 체크하는데, 한번에 하나의 child만이 컨넥션을 accept할 수 있다. 그 외에

호출되는 곳은 전역변수 값의 변경, 메모리 할당과 해제, 파일 locking 등이다.

MPM-worker에서는 각 child 프로세스의 리슨 소켓에 요청이 도착했는지 검사하기 위해 poll()을 사용한다. 그러므로, 정적/동적 문서의 비율에 상관없이 poll() 시스템 호출은 일정 비율을 차지한다. 또, descriptor의 정보를 조작하는 sys_fcntl64() 시스템 호출도 문서의 비율에 상관없이 일정한 값을 유지하며 네트워크 소켓관련 함수의 비율도 많은 부분을 차지한다.

4. 결론

MPM-worker에서는 정적 문서의 요청이 많을 때는 select()와 스래드 함수의 오버헤드가 대부분을 차지하며 동적 문서의 요청이 많을 때는 read()와 소켓 함수 그리고 스래드 함수의 오버 헤드가 크다. 그러므로, 전체적으로 살펴볼 때 MPM-worker에서는 파일 시스템 오버헤드와 스래드 synchronization 오버헤드가 크다고 할 수 있다.

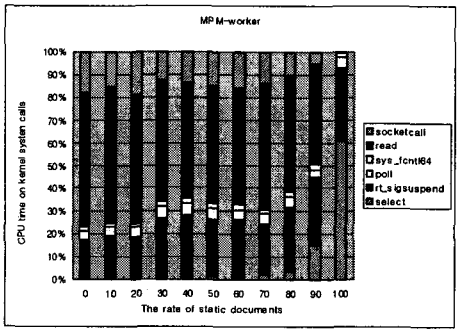

그림 3 MPM-worker의 시스템 호출 구성

참고문헌

[1]Yiming Hu, Ashwini Nanda, and Qing Yang, Measurement "Analysis and Performance Improvement of the Apache Web Server" In Proceedings of 18th IEEE International Performance, Computing and Communications Conference, 1999

[2]E. N. Nahum, Tsipora Barzilai, and Dilip Kandlur "Performance issues in WWW servers" In ACM Sigmetrics, 1999.

[3]Peter Druschel, Vivek S. Pai, andWilly Zwaenepoel "Extensible kernels are leading OS research astray" In SixthWorkshop on Hot Topics in Operating Systems, Cape Code, MA, May 1997.

[4]M. Frans Kaashoek, Dawson Engler, Gregory R. Ganger, Hector Briceno, Russell Hunt, David Mazieres, Tom Pinckney, Robert Grimm, John Janotti, and Kenneth Mackenzie "Application performance and flexibility on exokernel systems" In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, Saint-Malo, France, October 1997.

[5]M. Frans Kaashoek, Dawson Engler, Gregory R. Ganger, and Deborah A. Wallach "Server operating systems" In 1996 SIGOPS European Workshop, Connemara, Ireland, September 1996.

[6]Philippe Joubert, Robert King, Richard Neves, Mark Russinovich, John Tracey " High-Performance Memory-Based Web Servers: Kernel and User Space Performance" In Proceedings of the Annual Technical Conference2001 USENIX Annual Technical Conference, June 2001.

[7]http://www.opersys.com/LTT/