Readahead 기능을 활용한 NAND Flash 읽기 성능 향상에 대한 연구

박호준*, 임채덕**
*한국전자통신연구원
e-mail : hjpark1989@etri.re.kr, cdlim@etri.re.kr

A Study on Improvement Read Performance of NAND Flash based on Readahead Function

Hojoon Park, Chaeduk Lim
ETRI(Electronics and Telecommunications Research Institute)

요 약

임베디드 리눅스의 부팅은 일반적인 경우 Boot Loader, Kernel, Script 로 구성된 초기화 과정, Application Program 의 순서로 이루어진다. 이 경우 부팅 시간은 Power On 에서 최종 Application Program 이 동작을 시작하는 시점까지이다. 따라서 부팅 시간을 줄이는 방법은 부팅 과정의 중간 과정 중 불필요한 과정을 없애거나, 최적화하여 최종 단계에 빠르게 도착하게 만드는 것이다. 이러한 과정들에는 파일시스템 내의 데이터들을 메인 메모리로 복사하는 과정이 포함된다. 임베디드 시스템 내의 파일시스템은 주로 플래시 메모리에 저장되며, 플래시 메모리는 상대적으로 느린 속도로 동작된다. 따라서 부팅 시간은 상당히 많은 부분을 플래시 메모리에서 데이터를 복사하는데 사용된다. 결과적으로 부팅 시간을 줄이는 여러 방법들 중 flash-to-memory copy 의 시간을 줄이는 것은 효율 좋은 방법일 수 있다. 본 논문에서는 임베디드 시스템에 탑재되어 있는 플래시 메모리에서 메모리에 복사시 readahead 를 이용하여 복사시간을 효율화하는 방법을 제안한다.

1. 서론

리눅스가 임베디드 시스템에 탑재된 시기는 1990 년대 중후반이다. 당시 x86 architecture 기반의 PC 에서 동작되던 리눅스를 임베디드 시스템으로 포팅하면서 부팅 과정 역시 PC 의 그것을 그대로 가져온 것이 현재의 임베디드 리눅스의 부팅 방식이다[1].

임베디드 리눅스의 부팅 방식은 임베디드 시스템에 따라 약간 차이가 있지만 Boot Loader, Kernel, 초기화 스크립트, 응용 프로그램의 순으로 이어진다[2]. Boot Loader 는 최초 전원 인가에서 하드웨어 초기화, OS 의 핵심인 kernel 을 메모리로 복사하는 일을 수행한다. 이후 메모리에 복사된 kernel 은 초기화 과정을 거쳐서 초기화 스크립트 과정으로 진입한다. 초기화 스크립트 과정에서는 응용 프로그램을 실행하기 위한 다양한 환경 설정 등이 포함된다[3]. 마지막 절차로 응용 프로그램들이 수행된다.

임베디드 리눅스가 작동될 수 있는 통상적인 임베디드 시스템에서는 PC 에서처럼 디스크 장치를 쓰기 어려울 수 있으므로, 플래시 메모리를 파일시스템으로 활용한다[4]. 이러한 환경에서는 다른 컴퓨터 시스템과 마찬가지로 kernel 을 포함한 어떠한 응용 프로그램도 파일 시스템 자체에서는 실행될 수 없다. 따라서 응용 프로그램을 실행시키려면 파일시스템 안의 데이터를 메모리로 복사하는 과정이 필수이다[5]. 또한 플래시 메모리는 임베디드 리눅스가 작동되는 환경에서 접근 속도가 느린 장치에 해당한다. 그러므로 임베디드 리눅스의 부팅 속도를 결정하는 중요한 요인 중 하나가 플래시 메모리의 접근 속도이다[6].

또한, 플래시 메모리는 read/write 과정에 있어서 디스크 시스템과는 다르게 CPU 의존적이다. 디스크 시스템은 자체 컨트롤러가 내장되어 있어 DMA 와 유사한 작동 방식으로 특정 위치의 데이터를 요청(read/write)한 후 디스크 컨트롤러가 해당 작업을 완료할 때까지 CPU 는 다른 일을 수행할 수 있다. 그러나 플래시 메모리를 사용하는 임베디드 시스템에서는 하드웨어 단순화나 단가 문제로 인해 이러한 고급 컨트롤러를 사용하지 않고 CPU 가 직접 접근 요청을 내리고 일정 시간 대기한 다음, 데이터를 원하는 위치에 read/write 한다[7].

이러한 작업은 CPU 를 바쁘게 만들고 특정 단위 블록을 처리하는 동안 다른 일을 할 수 없도록 만든다. 게다가 CPU 가 다른 일을 하면 플래시 메모리로의 접근 요청은 지연된다.

결과적으로, 데이터를 파일 시스템에서 read/write 하려면 가급적 read/write 중 다른 작업은 대기시키거나 하지 말아야 한다.

본 논문은 리눅스 kernel 의 고유 기능인 readahead[8]를 이용하여 파일시스템 접근을 효율화하여 부팅 속도를 개선하는 방법을 제안한다.

2. 관련 연구

리눅스 kernel 의 고유기능인 readahead 는 인자로 주어진 파일시스템 내의 파일을 읽어서 메모리로 복사하는 일을 수행한다. readahead 의 기능이 보편적인 read 작업과 다른 점은 readahead 로 읽은 데이터가 메모리에 상주한다는 점이다. 이에 반해 일반적인 read 작업은 하나의 프로세스에 국한되므로 더 이상 해당 파일을 참조(open)하는 프로세스가 없다면 해당 프로세스의 종료와 함께 읽은 작업은 메모리에서 삭제된다. 그러나 readahead 는 readahead 를 사용한 프로세스가 종료되어도 OOM(Out Of Memory) 상태가 아닌 한 지속적으로 메모리에 상주된다. 따라서 이후 사용될 특정 파일들을 "미리" 읽어 두는 것이 가능하다.

이 기능을 활용하면 특정 프로세스가 원하는 파일이 미리 메모리에 탑재되어 있으므로, 향후 해당 프로세스가 원하는 파일을 오픈하여 read 작업 수행 시 파일시스템으로부터 읽어 들이질 않고 이미 읽은 것으로 간주되어 즉시 결과를 회신한다. 이는 해당 프로세스뿐만 아니라 해당 파일을 필요로 하는 모든 프로세스들에게 동일하게 적용되므로 읽기 작업 최적화가 가능하다.

readahead 는 원래 PC 환경에서 GUI 환경을 위해 사용된 바 있다. GUI 환경을 구동하기 위해서는 매우 많은 양의 파일들이 필요하고 매번 로그인/아웃을 반복할 때마다 이러한 파일들을 읽어 들이기 때문에 readahead 를 이용하여 로딩 속도 향상을 이룰 수 있다.

그러나 임베디드 리눅스에서는 제한된 메모리 양과 상대적으로 느린 읽기 성능을 가진 플래시 메모리 때문에 GUI 의 용도로 사용되기 보다는 용도가 한정된 응용 프로그램과 그 응용 프로그램이 필요한 파일들을 미리 읽기 위해 사용된다.

리눅스의 응용 프로그램은 파일 시스템에 저장된 실행 파일 자체와 일련의 공유가 가능한 라이브러리 파일들로 구성된다. 또한 응용프로그램 실행시 응용 프로그램의 의도에 따라서 읽혀 들어지는 파일들도 있을 수 있다.

이런 파일들이 readahead 의 대상이 되며 해당 응용 프로그램을 실행시키기 전에 먼저 사용한다.

3. Readahead 를 이용한 Flash 읽기 성능 시험

그림 1 은 응용 프로그램이 파일 시스템으로부터 메모리로 로드될 때의 과정을 표현한 것이다. 운영체제의 특성상 새로운 프로세스는 기존의 프로세스를 복사하여 생성된다.(fork) 새로운 프로세스는 exec 를 호출함으로써 파일 시스템 내의 실행 파일을 메모리로 복사한다. 새로운 응용 프로그램 실행 직후에는 프로그램을 유지하는데 필요한 공유 라이브러리가 복사된다. 이후 본격적으로 응용 프로그램 내의 메인 프로그램이 동작된다.

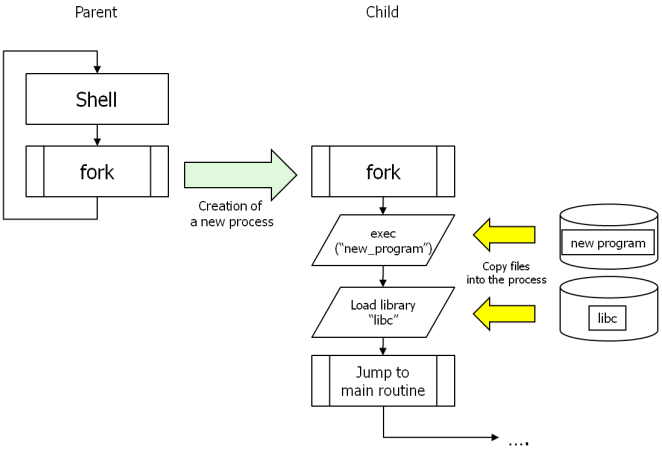

그림 1 응용 프로그램의 로딩

전술하였듯이 임베디드 시스템에 탑재된 플래시 메모리에서 데이터를 읽으려면 CPU 가 직접 개입을 해야 한다. 따라서 응용 프로그램이 파일 시스템에서 로드될 때에도 이와 같은 규칙이 적용된다. 임베디드 리눅스의 멀티 태스킹 환경 내에서 응용 프로그램은 2 개 이상이 동시에 작동될 수 있다. 그러므로, 공유 라이브러리를 필요로 하여 File I/O 를 요청 중인 응용 프로그램은 다른 응용 프로그램에 의해 선점되어 요청된 공유 라이브러리를 읽어 오는 작업이 지연된다.

그림 2 는 응용 프로그램 1 개가 이미 수행중이고 다른 응용 프로그램이 방금 실행되었을 때를 표시한 time table 이다. 그림 2 에서 App1 은 이미 수행중인 응용 프로그램이고 App2 는 방금 실행되어 공유라이브러리 읽기를 요청하고 있는 상태이다. 이때 App2 가 read 요청시 read system call 이 사용되고 스케줄러에 의해 바로 read service 로 가지 못하고 task switching 되어 CPU 는 App1 을 수행하게 된다. 이때 App2 의 time slice 가 만료되거나, App2 가 I/O 등의 이유로 CPU 선점을 포기해야만 다시 read 의 기회가 생긴다. 이제 App1 과 App2 가 모두 대기 상태에 다다랐으므로 CPU 를 선점(preempt)한 kernel 은 실제 read 작업을 수행할 수 있다. kernel 이 read 작업을 수행하는 동안 App2 를 포함한 다른 프로세스들은 수행할 수 없는 상태가 되고, 이 read 작업이 끝나야 App2 는 task switching 을 통해 CPU 를 선점하고 다음 작업을 수행하게 된다.

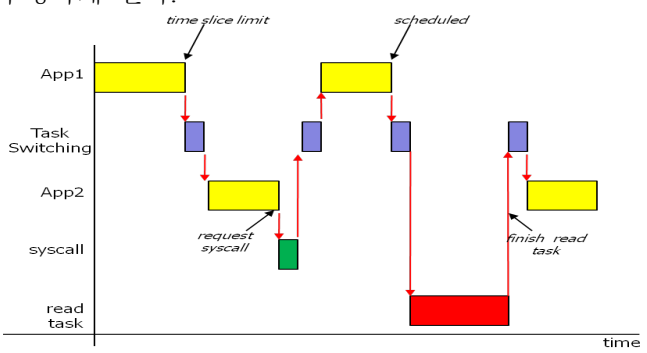

그림 2 readahead 가 없는 작업의 Time Table

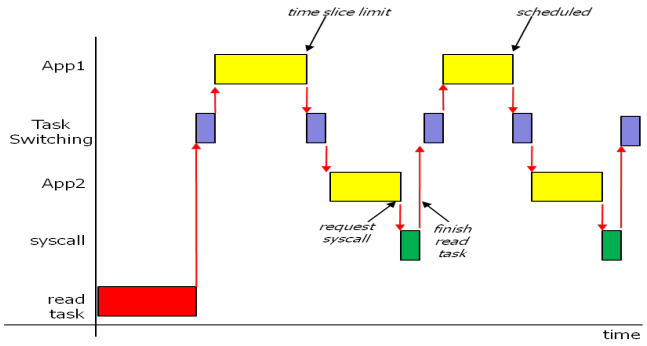

그림 3 readahead 가 적용된 Time Table

그림 3 은 App2 가 사용하는 공유 라이브러리를 readahead 를 이용해 "미리" 읽어 두었을 때의 동작이다. 그림 2 와 비교하여 read 작업시 소요된 task switching 시간도 같이 제거된다. 그러므로 그림 3 에서의 read 작업 완료시점이 그림 2 보다 먼저 끝나게 된다. 그림 2 에서의 task switching 시간 낭비는 read 작업이 빈번히 일어날수록 많이 발생되며, 동작 프로세스가 점점 많아질 수록 그 수는 기하급수적으로 증가한다. 또한 task switching 비용은 성능이 좋지 않은 시스템일수록 더욱 더 많이 증가된다.

따라서 임베디드 리눅스 부팅시 readahead 를 통한 "미리" 읽기 작업을 multi-tasking 이 시작되기 이전에 완료하면 응용 프로그램이 File I/O 를 요청할 때 이미 읽은 것으로 간주되어 task switching 비용 및 효율화가 이루어지고, 이것의 결과로 부팅 시간 단축으로 이어진다.

따라서, readahead 를 사용했을 때와 그렇지 않을 때의 부팅 시간을 비교하기 위해서 그림 4 와 같은 형태의 구조를 제안한다. 읽을 파일을 공유 라이브러리를 선택하지 않고 일반 파일을 선택한 것은 파일의 용량을 제어하기 쉽고 임의로 블록 크기를 설정하기 쉽기 때문이다. 또한 시간 차이를 명확히 하기 위해 미리 다른 응용 프로그램을 실행시켜 둔 상태이다. 해당 응용 프로그램은 CPU 점유율을 최대까지 올리는(boost up) 역할을 수행하는 것으로 task switching 이 자주 일어날 수 있는 환경을 조성한다.

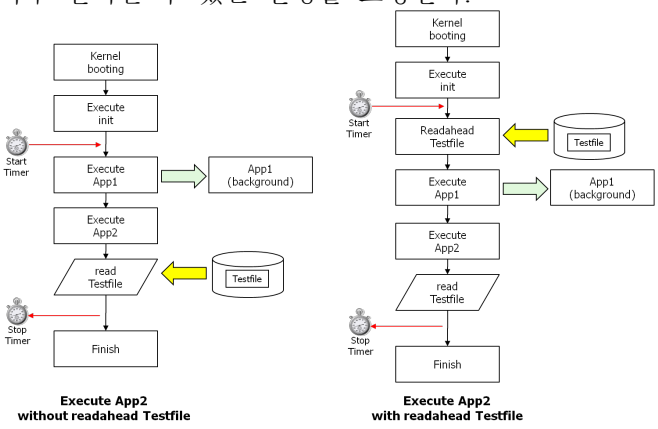

그림 4 성능을 비교하기 위한 Test Bed

따라서 읽혀 들어지는 파일들도 있을 수 있다.

본 논문에서 활용한 Hardware 는 삼성의 S3C6400 을 이용한 개발 보드(development kit)인 SMDK6400 이다. 이 보드(board)는 ARM11 core 를 이용한 보드로 533MHz clock speed 를 가진다. 메모리는 DDR2 type 으로 128MB 를 내장하고 있다. 플래시 메모리는 SLC type 으로 64MB 를 교체(replaceable) 할 수 있는 형식이다.

이 보드에 리눅스 version 2.6.21.5 를 이용하여 실험을 진행한다. 해당 보드에서 플래시 메모리의 데이터를 메모리로 복사할 때 약 2MB/sec 정도의 속도를 가진다.

리눅스는 page cache 를 File I/O 에 적극 사용을 하기 때문에 한번 실험을 하면 재부팅(reboot)을 하여 page cache 에 저장된 잔여 데이터들이 실험에 영향을 주지 못하도록 한다.

시간 측정은 linux kernel booting 이 종료되고 init 시작 후 시작 문자열(start string)인 "AAA"를 콘솔로 print out 한 후 응용 프로그램이 필요한 파일을 모두 읽은 직후 종료 문자열(finish string)인 "BBB"를 콘솔로 print out 한다. 콘솔은 개발용 host PC 에 연결되어 있고, 이것은 시작 문자열을 입력 받은 직후 시간을 카운트하여 종료 문자열을 보드로부터 입력 받을 때 경과된 시간을 10^{-2} 초 단위로 출력한다. 보드에서 직접적으로(directly) 사용자 입력을 하지 않은 것은 측정 시간의 정확도 때문이다.

보드는 kernel booting 직후 지정된 init 프로그램을 수행하게 되며, 해당 init 은 1 번 process 로 대부모(super parent) process 이다. 원칙적으로 정상적인 부팅 과정은 init 이후 inittab 파일 기술에 의해 부팅을 진행하나 본 논문에서는 실험을 위하여 init 은 단지 테스트 응용프로그램을 부르거나, readahead 를 이용해 파일을 "미리" 읽기를 하고 테스트 응용프로그램을 부르고 종료한다.

이때 테스트 응용 프로그램이 읽어들일 파일의 크기는 1MB 에서 500Kbyte 씩 증가하여 4MByte 까지 총 7 단계로 구성된다.

4. 시험 결과

그림 5 는 파일 크기 1MB 에서부터 4MByte 까지 읽기를 readahead 를 이용하여 "미리" 읽었을 때와 그렇지 않았을 때의 시간을 측정한 결과를 그래프로 나타낸 것이다.

읽어 들이는 파일의 크기가 1MB 일 때의 양쪽의 차이는 0.29 초이나, 읽어 들이는 파일의 크기가 증가할 경우, 그것과 비례하여 4MB 일 경우에는 1.65 초의 차이가 나타난다.

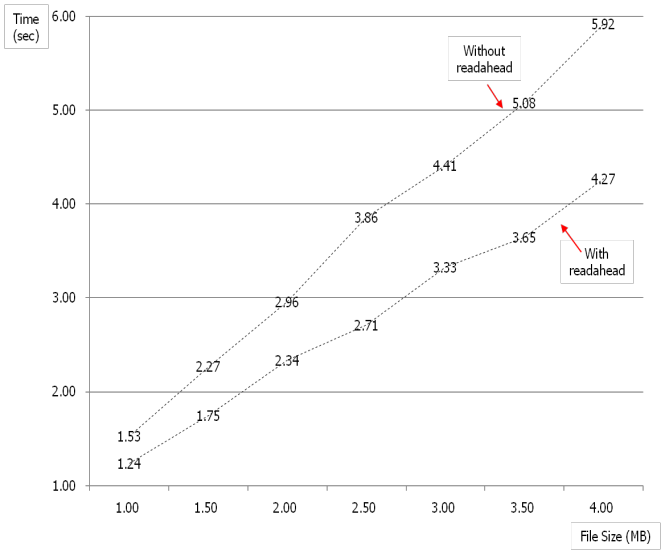

그림 5 성능 비교 도표

5. 결론

readahead 를 이용하여 실질적인 File I/O 와 Multi-tasking 이 일어나는 시점을 달리 하면 부팅 속도가 증가하는 것을 관찰할 수 있다. CPU 직접 개입의 플래시 메모리 접근은 전술하였듯이 Multi-tasking 환경에서는 적절치 못하므로, readahead 기능을 이용하여 읽기 작업을 선행하는 것이 성능 향상이 도움이 된다.

그러나 readahead 를 이용한 부팅 기술은 메모리 용량에 제한을 받는다. 메모리 용량이 작은 시스템인 경우는 오히려 page cache 가 저장될 공간이 부족하여 일찍(early) "미리" 읽었던 데이터들은 OOM 정책에 의해 메모리에서 삭제되므로 불필요한 행위를 초래한다.

이런 일련의 작업들은 제품 개발 시 반드시 고려되어야 하며, 반드시 필요한 파일들만을 readahead 하는 것이 중요하다.

참고문헌

[1] Christopher Hallinan, "Embedded Linux Primer, A Practical, Real-World Approach", Prentice Hall Professional Reference, pp.137

[2] Tim R. Bird, "Methods to Improve Bootup Time in Linux", Proceedings of the Linux Symposium Volume One, 2004

[3] Kenneth H. Rosen (1999). UNIX: The Complete Reference. McGraw-Hill Professional

[4] Atsuo Kawaguchi, Shingo Nishioka, Hiroshi Motoa, "A Flash-memory based file system", Proceedings of the USENIX 1995 Technical Conference, pp 13

[5] Wilshire, Phil. "eXecute In Place (XIP) overview", uCdot, August 28, 2002. Accessed September 25, 2007.

[6] Brett Glass, "There in a Flash: Flash Memory for Embedded Systems". http://www.embedded.com/98/9801.spec.htm

[7] K9F1208U0B, 64Mx8Bit NAND Flash Memory, Samsung Electronics, 2004

[8] Hojoon Park, "Fast Booting of Embedded Linux", CELF 3rd Korean Technical Jamboree, July 2008