Hanbit eBook

Realtime 65

Thinking About C/C++

프로그램 실행 환경편

프로그래머가 몰랐던 프로그램의 동작 원리

박수현 지음

한빛미디어 Hanbit Media, Inc.

Thinking About

C/C++

프로그래머가 몰랐던 프로그램의 동작 원리

Thinking About C/C++: 프로그래머가 몰랐던 프로그램의 동작 원리 프로그램 실행 환경편

초판발행 2014년 05월 15일

지은이 박수현 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-711-8 15000 / **정가** 9,900원

책임편집 배용석 / **기획·편집** 정지연
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 최송실
영업 김형진, 김진불, 조유미 / **마케팅** 박상용, 서은옥, 김옥현

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea

저자 소개

지은이_박수현

홍익대학교에서 컴퓨터공학 학사부터 박사까지 마쳤으며, 현재 현대오토에버에 재직 중이다. 약 12년에 걸친 홍대 생활로 잘 놀 것 같다는 오해를 자주 받고 있다. 사실 홍대 앞 번화가에 대해서는 잘 모르지만, 홍대 근처 어느 집에서 자장면을 시켜야 맛있는지는 조언해 줄 수 있다. 운영체제, 시스템 프로그래밍에 관심이 많다.

저자 서문

중고등학교 시절 나를 끊임없이 괴롭히던 한 가지 생각은 “이 많은 것들을 배워서 도대체 어디에 써먹는 것일까”였다. 물리학자도, 수학자도 될 생각이 없었기에 미적분은 그저 성적 유지를 위해서 공부하는 것에 불과했다. 대학교에 진학하고 컴퓨터 공학을 전공하면서도 이 생각은 그대로였다. 자료구조와 알고리즘 수업을 듣는다고 해서 멋있는 게임을 만들 수도 없었고, 컴퓨터 구조나 운영체제 수업을 듣는다고 임베디드 장치를 만들 수 없었기에 그 허탈감은 이루 말할 수 없었다.

대학원 진학을 고민하던 시절, 술자리에서 한 선배가 필자에게 질문을 던졌다. “넌 명색이 컴퓨터 공학과 졸업자인데 컴퓨터가 어떻게 부팅되는지 정확하게 설명할 수 있느냐”라고 말이다. 망치로 머리를 얻어맞은 느낌이었다. 비록 모든 내용을 다 배우지는 못했지만, 4년 동안 컴퓨터 공학 전공 수업을 들으면서 한 부분씩은 배웠을 텐데 이들이 어떻게 연결되고 어떻게 동작하는지 이해하려는 시도조차 하지 않았다. 수동적으로 배우기만 하고 내가 만든 프로그램이 어떻게 동작하는지, 무엇이 문제인지 고민조차 해 보지 않은 것이다.

대학원에 진학하고 6년이 넘도록 컴퓨터 시스템에 관련된 공부를 하면서 가장 많이 받은 질문은 “왜 인기도 없는 시스템을 공부하느냐”였다. 답은 간단했다. 어떤 언어를 쓰고 어떤 프로그램을 만들고 어떤 운영체제를 사용하든지 그 근간은 시스템이라고 생각하기 때문이다. 아무리 능숙한 프로그래머라 할지라도 언어의 기본적인 개념을 이해하지 못하거나 프로그램이 동작하는 근본 원리나 플랫폼의 세부사항을 알지 못한다면 이는 능숙한 ‘코드 작성자’밖에 안 된다. 실무를 통해 겪었든 수업이나 책을 통해서 공부했든 자신이 몸담은 분야에 대한 기반 지식이 없다면 살아남기 힘들다고 생각한다. 기반 지식보다 더 문제인 것은 이미 배운 내용을 등한

시하고 사용하지 않는다는 점이다. 아마도 이는 필자가 느낀 것과 마찬가지로 이들을 어디에 써먹어야 할지, 왜 필요한지를 이해하지 못하기 때문이라고 생각한다.

이 책은 그래서 쓰기 시작했다. 컴퓨터공학을 전공하지 않았거나 잘 모르는 사람이 보기에는 조금 어려울 수도 있고, 실무 경험이 많거나 컴퓨터에 대한 지식이 많은 사람이 보기에는 너무 쉬울지도 모른다. 책을 쓰는 데 많은 도움을 주신 분의 평가처럼 대상 독자층이 참으로 '애매모호'하다.

하지만 나는 이 책을 통해서 여러분이 배웠을지도 모르는 그 지식이 어떻게 사용될 수 있으며 왜 중요한지 꼭 이야기하고 싶었다. 한 줄의 코드 라인에 캐시의 철학을 담을 수 있는 사람이 진정한 프로그래머라고 생각한다. 그리고 이 책을 읽는 많은 분은 캐시가 무엇인지 그리고 어떻게 프로그램을 만드는 것인지 이미 다 알고 있다고 생각한다. 다만 왜 캐시가 중요하고, 캐시가 어떻게 프로그램에 영향을 미치는지 그리고 작성하는 코드가 캐시의 동작에 어떤 의미를 가지는지를 모를 뿐이라고 생각한다. 비단 캐시뿐만 아니라 이 책에서 설명하는 모든 부분은 익히 아는 내용일 것이다. 다만 이들이 여러분의 프로그램과 어떤 연관이 있는지 모를 뿐이다.

사실 이 책에서 이야기하는 내용은 몰라도 회사에서 일하고 프로그램을 만들고 돈을 버는 데에는 아무 지장이 없다. 다만 이 책을 다 읽고 덮을 때 즈음 여러분이 프로그램의 소스 코드뿐 아니라 그 외에 많은 부분을 둘러볼 수 있는 좀 더 넓은 시야를 갖게 되길 바란다.

마지막으로 이 책이 나오기까지 많은 도움을 주신 모든 분께 감사의 말을 전한다.

대상 독자 및 참고사항

초급 | 초중급 | **중급** | 중고급 | 고급

이 책은 C/C++ 등 프로그래밍 언어에 어느 정도 익숙하며 프로그램을 만들고 실행해 본 경험이 있는 중급 이상의 프로그래머를 대상으로 한다.

이 책에서는 주로 POSIX를 따르는 Unix 계열의 운영체제와 C/C++ 언어를 주로 다루고 있다. 이 책에 수록된 예제는 대부분 이해를 도우려고 작성된 슈도 코드 또는 스켈레톤 코드 형태이며 상당수가 불완전하다. 각 코드의 실행 환경은 대부분 설명 부분에 기재하였으나, 없는 경우 리눅스와 GCC를 사용하였다고 보아도 무방하다.

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.

   500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 보다 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. 무료로 업데이트되는 전자책 전용 서비스입니다.

   종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위해 DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

1 | I/O

사용하는 디스크만 해도 그 종류가 다양하다. SD도 있고 플래시 메모리도 있고 일반적인 자기 디스크도 있다. CD나 DVD도 디스크이며 심지어 테이프와 같은 장치도 디스크라 할 수 있다. 스마트폰의 SD 카드 또한 디스크라 할 수 있겠다. 엄밀히 플래시 메모리나 SSD, SD 카드 등을 디스크라 부르긴 힘들지만, 여기 있는 파일들 역시 Open이나 OpenFile 등의 함수를 사용해서 접근하는 것은 매한가지다. 그리고 FAT16, FAT32, NTFS, EXT3, EXT4 등 수많은 종류의 파일 시스템이 존재한다.

디스크가 어떤 종류인지 그리고 어떤 파일 시스템을 사용하는지에 따라 파일을 찾고, 읽고, 쓰는 방법이 제각각 다를 텐데 어째서 우리는 Open만 쓰면 되는 것일까? 그렇게 해도 아무런 문제가 없을까? 이제껏 아무런 문제 없이 잘 사용하고 있었는데 왜 이제서야 I/O가 어떤 특성을 가지는지 알아야 하는 것일까? 이제부터 그 이유를 알아보자.

[코드 1-1]

```
int main()
{
   printf("Hello, World!\n");
   return 0;
}
```

무슨 설명이 필요한가! 아마 많은 프로그래머가 가장 처음 컴파일하고 실행한 코드가 바로 이 코드가 아닐까 싶다. 혹자는 'Hello, World' 프로그램만 작성할 줄 알아도 프로그램의 모든 것을 아는 것이라고도 할 정도니 말이다. 단순히 화면에 'Hello, World!' 문자열만 출력하는 프로그램일 뿐이지만, 모니터에서 이 문자열

을 보고 기쁨의 눈물을 흘린 적도 있다. 믿거나 말거나!

컴파일러를 설명하면서 잠깐 이야기했지만, 컴파일러뿐 아니라 링커나 로더, 운영체제, 라이브러리 등 수많은 요소들로 인하여 프로그래밍이 점점 더 쉬워지긴 하였다. 사실 먼 옛날(?)에만 해도 화면에 문자열을 출력하려면 수많은 작업을 거쳐야만 했다. 모니터의 해상도나 규격, VGA 카드의 종류, 폰트, 문자셋, 심지어 한글이 조합형인지 완성형인지를 따져봐야 하는 시절도 있었다. 지금도 모든 과정이 자동으로 해결되는 것은 아니지만, 그래도 대부분은 하드웨어나 운영체제에 관련된 요소를 고려할 필요 없이 적당한 라이브러리를 가져다 쓰면 화면에 원하는 내용이 출력되니 참으로 편하지 않을 수 없다.

아마 여러분도 파일을 읽거나 쓰는 등의 코드를 작성하면서 디스크의 용량이 얼마이고 어떻게 구성되어 있는지, 제조사가 어딘지, 어떤 운영체제를 쓰고 어떤 파일 시스템을 쓰는지 한 번도 생각해 본 적이 없을 것이다. 파일을 열고 싶으면 그냥 Open( )이나 OpenFile( ) 등의 함수를 쓰기만 하면 그만이다. 파일 경로나 크기, 권한 등이야 물론 프로그래머가 신경 쓰고 관리해야 하는 부분이지만, 그 외에는 안중에도 없는 것이다.

1.1 I/O 처리는 누가 담당하는가

I/O라는 딱지를 붙일만한 부분은 아주 많다. 디스크, 모니터, 키보드, 마우스, 프린터, 기타 수만 가지 주변 장치와 더불어 네트워크도 I/O의 범주에 포함할 수 있다. 쉽게 말해서 CPU와 메모리를 빼면 모두 I/O라고 봐도 무방하다. 이 책이 하드웨어 유지 보수에 관한 설명서가 아닌 이상 모든 종류의 주변 장치를 다 다룰 수는 없으므로 디스크, 모니터, 키보드, 네트워크와 같이 가장 일상적으로 사용되는 I/O만 짚어 보도록 하겠다.

우선 이런 I/O 장치들은 공통적인 특성이 있다. 바로 디바이스 드라이버라는 존재

다. 윈도우를 쓰면서 새로운 장치를 사용할 때 설치해야 하는 바로 그 드라이버를 말한다. 디바이스 드라이버는 운영체제, 아키텍처를 막론하고 제공되어야 한다. 드라이버는 쉽게 말하면 해당 장치를 사용하기 위한 매뉴얼이라고 볼 수 있다.

예를 들어, H모 사에서 제조하는 SUV 차량과 S모 사에서 제조하는 트럭은 세부 구조가 다르고 조작하는 방법도 다르지만, 보통 운전면허를 가진 사람이라면 알 수 있는 '핸들 조작', '가속', '정지' 등의 기능은 다 있을 것이다. 다만, 그 조작 방법이나 구현이 약간씩 달라서 SUV 차량에서 핸들 조작을 하려면 이러이러해야 한다라고 기술한 것이 매뉴얼이 되는 것이다.

매뉴얼은 이처럼 표준으로 제공하는 기능을 어떻게 사용해야 하는지에 대한 설명 외에도 차량이 제공하는 기타 기능에 대한 사용 방법 또한 명시한다. 디바이스 드라이버도 마찬가지로 일반적으로 제공하는 기능을 어떻게 사용하는지 그리고 해당 장치가 제공할 수 있는 부가 기능이 어떤 것이 있으며 어떻게 사용해야 하는지를 명시한다. 그리고 그 기능을 요청하였을 때 하드웨어를 실제로 조작하고 관리하는 것이 바로 디바이스 드라이버가 하는 일이다.

설명이 좀 어려운데, 예를 들어 새로운 USB 메모리 하나를 장만했다고 가정하자. 새로 산 USB 메모리를 처음 컴퓨터에 연결하면 대부분은 해당 장치의 드라이버를 검색해서 자동으로 설치하고 사용할 수 있도록 준비해준다. USB 메모리로 할 수 있는 일은 어떤 파일이나 폴더가 있는지 검색하고, 파일을 저장하고 지우는 등의 일이다. 그리고 그와 별개로 USB 메모리를 만든 제조 회사가 USB 자동 백업이나 암호화 등의 기능을 제공할 수도 있다. 이럴 때 디바이스 드라이버에는 다음 기능이 구현되어 있어야 한다.

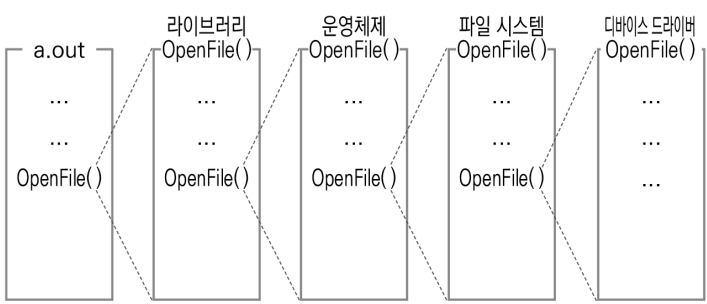
- 파일/폴더 목록 제공 및 검색
- 파일 열기
- 파일 쓰기

- 파일 삭제
- 백업
- 파일 암호화

USB 메모리에 있는 파일을 열기 위해서 프로그램을 작성했다고 가정하자. USB 메모리의 드라이브 문자가 'F'로 할당되고 가장 상위에 'test.txt'라는 파일이 저장되어 있다면 아마도 OpenFile("F:\test.txt"…)과 같은 방법으로 파일을 열 것이다. 이 함수는 운영체제나 혹은 개발 툴이 제공하는 표준 라이브러리에 포함되어 있으므로 사용하는 함수는 표준 라이브러리에 있는 함수를 호출하는 꼴이 된다. 그럼 라이브러리에 있는 'OpenFile'이라는 함수는 무슨 일을 할까?

이 함수는 운영체제에 파일의 경로와 권한 등의 정보를 넘겨주면서 일을 떠넘긴다. 운영체제는 경로를 통해서 어떤 디스크에 접근하는지 알아본 다음, 해당 드라이브가 사용하는 파일 시스템 정보를 얻어와서 파일 시스템이 제공하는 OpenFile을 호출하고, 이 함수는 다시 디바이스 드라이버가 제공하는 OpenFile 함수를 호출한다. 그림으로 보면 다음과 같다.

[그림 1-1] OpenFile 함수 호출 순서



디바이스 드라이버와 파일시스템, 운영체제 등은 레이어 구조로, 상위 구조에 추상화 기능을 제공한다.

사실 함수 이름이나 호출 순서, 호출 과정이 정확하게 일치하는 것은 아니지만, 간

단히 하면 [그림 1-1]과 비슷하다. 결국, 디바이스 드라이버는 일종의 API(Application Programming Interface)를 제공한다. 디바이스 드라이버를 사용하는 입장에서는 사용하려는 기능이 내부적으로 어떻게 작성되었는지 전혀 알 필요 없이 약속된 형태로 제공되는 기능을 사용하기만 하면 된다.

프로그래머 입장에서는 라이브러리가 API이고 라이브러리 입장에서는 운영체제의 함수[01]가 API 역할을 한다. 아무것도 몰라도 OpenFile이라는 함수만 쓰면 이런 함수들이 API로 제공되고, 내부 기능을 몰라도 프로그램 작성에 아무 영향을 미치지 않는다.

필자도 이런 API 내부가 어떻게 작성되어 있으며 어떻게 동작하는지 알지 못하고 관심도 없다. API는 대부분 시키는 일을 그대로 할 뿐이며 내부의 소스가 어떻게 되어있는지 알아도 별 필요가 없다. 주목해야 할 점은 API의 구현이 아니라, 실제 디스크와 같은 저장 장치가 어떤 식으로 동작하며 이런 특성에서 무엇을 골라서 쓸 수 있느냐다. 그 시작으로 가장 흔하게 사용하는 하드 디스크를 먼저 보자.

1.2 디스크

사실 하드 디스크라는 용어는 아주 먼 옛날(?) 사용되던 플로피 디스크보다 단단해서 그와 구분하기 위해 지은 이름이다. 플로피 디스크와 하드 디스크는 동작하는 방식이 거의 유사하다.

하드 디스크는 컴퓨터에서 몇 안 되는 기계 장치 중 하나다. 디스크 표면은 자기 물질로 덮여 있고 자기 물질로 기록된 데이터를 읽기 위해서 디스크가 회전하며 디스크 암이 움직여 헤드를 데이터가 기록된 위치로 옮긴다.

01 · 일반적으로 시스템 콜이라고 한다.

[그림 1-2] 하드 디스크 구조[02]

하드 디스크는 복잡한 기계 장치다.

하드 디스크를 구매해 보았다면, 하드 디스크의 성능을 표시하는 기준 중에 회전 속도Revolutions Per Minute, RPM가 표기된 것을 본 적이 있을 것이다. 이것은 디스크가 회전하는 속도를 나타내며, 회전 속도가 빠를수록 디스크 헤드가 데이터의 위치에 좀 더 빠르게 접근한다. 디스크에서 원하는 데이터의 위치를 찾아서 읽고 전송하는 데 걸리는 시간인 디스크 접근 시간은 다음과 같이 구할 수 있다.

> 디스크 접근 시간Access Time = 디스크 암이 적절한 디스크 트랙으로 이동하는 시간Seek Time + 디스크 헤더가 적절한 디스크 섹터에 위치할 때까지 디스크를 회전하는 시간Rotation Time + 데이터를 읽어서 보내는 시간Data Transfer Time

RAM이나 ROM 같은 메모리 방식의 저장 장치는 탐색 시간Seek Time이나 회전 시간Rotation Time은 많이 걸리지 않지만, 하드 디스크는 기계 장치이다 보니 디스크 헤드를 움직이는 시간이 더 소요된다. 잘 모르지만 아마 하드 디스크의 디바이스 드라이버에 있는 파일 읽기 함수에 파일의 위치로 디스크 헤드를 보내는 코드가 작성되어 있을 것이다.

---

02 Copyright 2013 Evan-Amos. 이 이미지는 크리에이티브 커먼즈 〈저작자표시-동일조건변경허락 3.0 Unported〉에 따라 이용할 수 있다.

이렇게 디스크 접근 시간에 디스크 헤드를 움직이는 시간이 포함되다 보니 같은 크기의 파일을 읽어도 그 시간이 제각각일 때가 많다. 파일이 조각나서 여기저기 저장되어 있을 수도 있고, 프로그램 혼자서만 디스크를 쓰는 것도 아니기 때문이다.[03] 상황이 이렇다 보니 윈도우 같은 운영체제는 조각 모음이라는 도구를 제공한다. 동일한 파일의 조각을 연속된 위치에 모으면 하나의 파일을 읽을 때 탐색 시간과 회전 시간이 최소화될 것이다. 따라서 파일을 읽어오는 시간이 줄어들고 성능이 향상된다. 기본적으로는 맞는 말이지만, 때에 따라서 순서대로, 차례차례, 차곡차곡 저장된 파일이 훨씬 나쁜 성능을 보일 수도 있다.

1.2.1 디스크 스케줄링

순서대로 파일이 저장되어 있을 때 언제 나쁜 성능을 보이는지 알아보자. 이 문제를 이해하기 위해서는 우선 디스크의 스케줄링에 대해서 이해해야 한다. 디스크는 앞에서 설명했듯이 지정된 데이터의 위치로 디스크 헤드를 옮기는 시간이 매우 중요해서 이 시간을 줄이기 위해 다양한 스케줄링 방법을 사용한다.

> **NOTE**
>
> 디스크가 데이터를 읽고 쓰는 것은 엘리베이터와 비슷하게 거리뿐만 아니라, 현재 디스크 암이 움직이는 방향 역시 중요한 요소가 된다. 5층에서 내려가기 위해서 엘리베이터 버튼을 눌렀을 때 거리는 3층에 올라오고 있는 엘리베이터가 더 가깝지만, 9층에서 내려오는 엘리베이터가 5층에 설 확률이 훨씬 높다. 만원이 아니라면 말이다.

가장 간단한 방법은 시간에 따라서 요청된 데이터의 위치로 바로바로 디스크 헤드를 옮기는 스케줄링이다. 디스크 헤드가 여기저기 움직이므로 당연히 최적화된 스케줄링 기법은 아니다.

효율적인 디스크 스케줄링 기법은 엘리베이터의 움직임을 생각하면 이해하기 쉽

03 『Thinking about C/C++ 〈프로그램 개발편〉(한빛미디어, 2014)』의 「3장 멀티 코어 시대의 C와 C++」 참고

다. 3층에 서서 위로 올라가기 위해 엘리베이터 버튼을 누를 때 엘리베이터가 정지해 있다면 3층으로 엘리베이터가 움직일 것이다. 엘리베이터가 내려가고 있다면 엘리베이터는 3층에 정지하지 않고 아래로 내려갔다가 올라올 때 3층에 정지한다. 누가 몇 층에서 먼저 버튼을 눌렀느냐는 보다는 엘리베이터가 진행하는 방향이 중요한 요소가 된다.

디스크 스케줄링도 이와 유사하다. 예를 들어 디스크에 데이터가 1부터 100까지 존재하며 순서대로 배치되어 있다고 가정하자. 디스크 헤드는 1에서 증가하는 방향으로 움직이거나 100에서 감소하는 방향으로 움직일 수 있다. 여러 프로그램이 디스크에 있는 데이터를 요청하는 데, 데이터 요청이 30, 50, 10, 20, 70, 40의 순서대로 들어왔다고 가정하자. 디스크 헤드가 현재 40에 있으며 증가하는 방향으로 움직이고 있다면 이 데이터 요청은 증가하는 방향에 따라서 정렬된다. 따라서 40, 50, 70의 데이터 요청을 우선 처리할 것이다.

디스크 헤드가 70까지 이동해서 데이터 요청을 처리하면 다음은 감소하는 방향으로 움직이게 되고 데이터 요청은 30, 20, 10의 순서로 처리한다. 감소하는 방향으로 처리하는 과정에서 80의 데이터 요청이 들어오더라도, 이미 감소하는 방향으로 디스크 헤드가 움직이기 시작하였으므로 해당 요청은 지연되고 나중에 처리된다. 엘리베이터의 움직임과 거의 같다고 볼 수 있다.

때로는 디스크 스케줄링 때문에 순서대로 배치된 데이터가 최악의 성능을 보이기도 한다. 프로그램이 데이터 1번부터 10번까지 처리하며 1번 데이터가 41의 위치에 있고, 이후 데이터가 50의 위치까지 순서대로 저장되어 있다고 가정해 보자.

[그림 1-3] 데이터 배치표

41	42	43	…	48	49	50
#1 데이터	#2 데이터	#3 데이터	…	#8 데이터	#9 데이터	#10 데이터

데이터가 오른쪽으로 순서대로 저장되어 있다.

프로그램은 시작하면서 디스크로부터 1~10번의 데이터를 요청한다. 1번 데이터가 도착하면 프로그램은 이 데이터를 처리하고 처리하는 중에도 디스크는 계속 데이터를 읽어서 전송하지만, 프로그램은 1번 데이터를 처리하느라 데이터를 전송받지 못한다. 디스크가 4번 데이터를 읽기 시작할 때가 되어서야 프로그램이 2번 데이터를 처리할 준비가 되지만, 디스크가 10번 데이터까지 모두 읽은 다음 다시 돌아갈 때가 되어야 2번 데이터를 다시 읽어올 수 있다. 따라서 디스크 헤드가 다시 2번 데이터 위치로 갈 때까지 프로그램은 아무 일도 못 하고 디스크가 그동안 읽은 데이터 역시 무용지물이 된다.

1.2.2 인터리빙

이럴 때 사용할 수 있는 방법은 무엇일까? 바로 인터리빙Interleaving이라는 기법이다. 용어는 거창하지만, 그 원리는 매우 간단하다. 연속적인 데이터를 연속적이지 않게 기록하는 것이다. [그림 1-3]의 데이터 순서를 약간 바꿔서 저장해 보자.

[그림 1-4] 인터리빙 기법에 의한 데이터 배치표

41	42	43	44	45	46	47	48	49	50
#1	#5	#9	#2	#6	#10	#3	#7	#8	#4

두 칸 간격으로 데이터가 저장되어 있다.

[그림 1-4]와 같이 데이터가 저장되었을 때 앞의 시나리오를 그대로 적용해 보자. 프로그램은 동일하게 1번 데이터를 전송받아서 처리하고 디스크는 계속 움직여서 2번 데이터 위치에 도달하면 프로그램은 1번 데이터의 처리가 끝난다. 데이터를 전송받을 준비가 된 프로그램은 디스크로부터 2번 데이터를 전달받아서 처리를 시작하고 그동안 디스크는 3번 데이터의 위치까지 읽어간다. 2번 데이터의 처리가 끝난 프로그램은 다시 3번 데이터를 전송받고 처리 작업을 시작한다. 언뜻 보면 이해가 되지 않는 동작 순서일 수 있으나, 이것은 데이터의 의존성이나 순서보다 데이터가 저장된 위치가 디스크 스케줄링에 절대적인 영향을 미치는 것을 보여준다.

사실 요즘 디스크들은 이런 인터리빙을 사용하지 않아도 될 만큼 충분한 디스크 버퍼를 제공한다. 그럼에도 인터리빙을 언급한 이유는 인터리빙이 디스크에만 사용되는 기법이 아니고 디스크 버퍼보다 훨씬 큰 데이터를 다루는 데 있어서 여전히 유용하기 때문이다. 또한, 인터리빙이 아니더라도 프로그램을 작성할 때 디스크에 데이터를 어떤 순서로 저장하는가는 프로그램의 성능을 좌우하는 중요한 요소가 되기 때문이다.

1.2.3 디스크 쓰기

좋은 디스크를 쓰면 해결되지 않을까? 사실 많은 경우에 맞는 말이다. 돈으로 해결되지 않는 문제는 그렇게 많지 않다. 디스크 성능이 좋지 않다면 더 좋은 디스크를 사용하거나 SSD를 쓰면 되고, 메모리를 확장하거나 CPU를 더 빠른 것으로 바꾸면 프로그램 또한 더 빨리 동작한다. 하지만 아무리 비싼 디스크를 사용해도 메모리나 CPU보다 동작 속도가 느리다는 사실은 변함없다. 디스크는 값싸고 용량 크고 느리다는 사실을 잘 알기에 많은 운영체제는 하드 디스크에 접근하는 횟수를 최대한 줄이려고 노력하며 어쩔 수 없이 하드 디스크를 사용하면 디스크 헤드가 최소로 움직이도록 한다. 이런 노력의 하나로 운영체제는 프로그램의 디스크 쓰기 요청을 모아서 한꺼번에 하드 디스크에 기록하기도 한다.

프로그래머들이 유념해야 할 것은 이런 특성 때문에 파일에 어떤 데이터 쓰기를 요청한다고 해서 그 즉시 파일에 기록되지 않는다는 점이다. 앞에서 설명한 것처럼 쓰기를 요청하면 운영체제나 시스템이 가진 특정 메모리 영역에 실제로 기록해야 하는 위치와 내용을 저장해 두었다가 시스템이 한가하거나 일정 시간이 되면 메모리에 저장된 내용을 한꺼번에 기록한다. 프로그램이 이런 시스템이나 운영체제의 디스크 버퍼를 사용하지 않고 싶다면 파일을 열 때 함수에 적당한 인자를 줄 수 있다. 예를 들어, 윈도우는 다음 방식으로 디스크 버퍼를 사용하지 않을 수 있다.

[코드 1-2]

```
      HANDLE fildHandle = CreateFile( filePath,
01       GENERIC_READ ¦ GENERIC_WRITE,
02       FILE_SHARE_READ ¦ FILE_SHARE_WRITE,
         NULL,
         CREATE_ALWAYS,
03       FILE_ATTRIBUTE_NORMAL ¦ FILE_FLAG_NO_BUFFERING ¦ ILE_FLAG_WRITE_THROUGH,
         NULL);
      WriteFile(fileHandle, data, fileSize, &dwWritten, NULL);
      ……
```

01 파일에 대한 권한을 보여준다.
02 파일이 없으면 권한으로 만든다.
03 버퍼링하지 않도록 지정한다.

유닉스 계열의 운영체제는 다음과 같이 디스크 버퍼를 사용하지 않도록 설정할 수 있다.

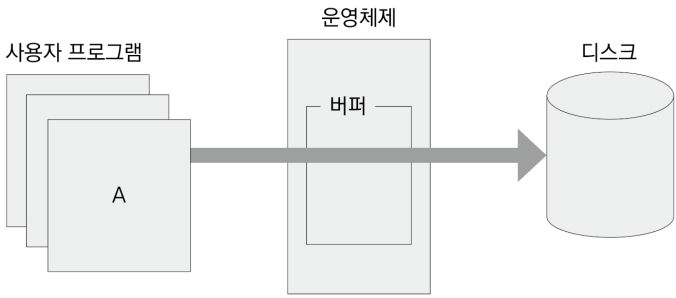
[코드 1-3]

```
01    int fd = open( filePath, O_RDWR ¦ O_DIRECT ¦ O_SYNC );
      write(fd, data, sizeof data);
      ……
```

01 파일 버퍼링을 사용하지 않도록 지정한다.

운영체제마다 사용하는 플래그 값이나 형태가 조금씩 다르지만, 대부분은 [코드 1-3]과 같이 캐시를 쓰지 않고 디스크에 그 내용이 전달될 때까지 쓰기 함수(WriteFile이나 write 함수)가 반환되지 않고 기다리도록 한다. 이렇게 열린 파일에 어떤 내용을 쓸 때는 운영체제나 시스템의 버퍼를 거치지 않고 바로 디스크로 전달된다.

[그림 1-5] 파일 버퍼링을 사용하지 않을 때 프로그램 흐름도



메모리를 거치지 않고 데이터가 디스크로 바로 전달된다.

하지만 불행히도 이런 설정으로 사용자 프로그램에서 데이터가 직접 디스크에 전달된다고 해도 여전히 디스크에 기록되지 않을 가능성이 존재한다. 앞서 디바이스 드라이버를 잠깐 설명하였지만, 디스크에 데이터를 쓰도록 요청하면 이것은 디스크의 디바이스 드라이버에서 제공하는 쓰기 함수를 호출하는 것과 다르지 않다.

디바이스 드라이버의 쓰기 함수가 디스크에 실제 기록하지 않고 디스크의 버퍼에 데이터를 기록했다가 몰아서 한꺼번에 디스크에 쓴다면 정전 등의 이유로 이 데이터가 소실될 위험이 여전히 존재하게 된다. 특히나 대용량 데이터는 부분적으로 데이터가 소실될 가능성도 매우 크다. 프로그래머 입장에서 데이터가 반드시 디스크에 기록되었다는 것을 보장받을 방법은 많지 않다. 특수한 디스크 장비를 사용해서 디스크에 데이터를 기록할 때는 모든 작업을 중단하고 쓰기 작업을 완료한 뒤에 다른 일을 하도록 강제한다면 모를까 모든 경우에 이런 디스크 장비를 쓸 수도 없는 노릇이다.

가장 일반적이면서도 유용한 방법은 디스크에 데이터를 기록하는 것이 신뢰성이 낮으므로 저장된 데이터를 사용하기 전에 해당 데이터가 문제가 없는지 체크섬 Checksum[04] 등을 통해서 점검하는 것이다. 이 방법으로 데이터의 손실을 막을 수는

04 · 체크섬이란 어떤 데이터에 대한 해시(Hash) 값으로, 데이터가 올바른지 간단하게 검사하는 용도로 사용된다.

없지만, 적어도 잘못된 데이터를 처리하는 일은 상당 부분 막을 수 있다.

1.2.4 페이징

디스크를 사용하는 주된 이유의 하나는 바로 가상 메모리다. 페이징Paging은 대표적인 가상 메모리 관리 기법으로 컴퓨터의 메모리 공간을 일정 크기(페이지 단위)[05]로 나누어 관리하며, 지금 당장 사용하지 않는 메모리 공간은 디스크와 같이 좀 더 큰 공간에 저장하고 사용한다. 일반적으로 컴퓨터가 사용할 수 있는 메모리 공간은 사용하는 비트의 크기에 따라 결정되는데, 예를 들어 32비트 CPU는 사용할 수 있는 최대 주소 공간이 2의 32승까지다. 당연히 64비트 CPU는 2의 64승만큼 사용할 수 있다.

요즘은 메모리 가격이 싸져서 4GB 정도의 메모리는 쉽게 확보하지만, 2의 64승만큼의 메모리를 사용하는 경우는 드물다. 페이징은 이렇게 실제 메모리 크기가 사용할 수 있는 메모리 공간보다 작을 때 하드 디스크와 같은 외부 저장 장치를 가상의 메모리처럼 사용할 수 있게 해준다. 간단히 설명하면 하드 디스크를 메모리, 그리고 메모리를 CPU 캐시로 사용한다고 할 수 있다.

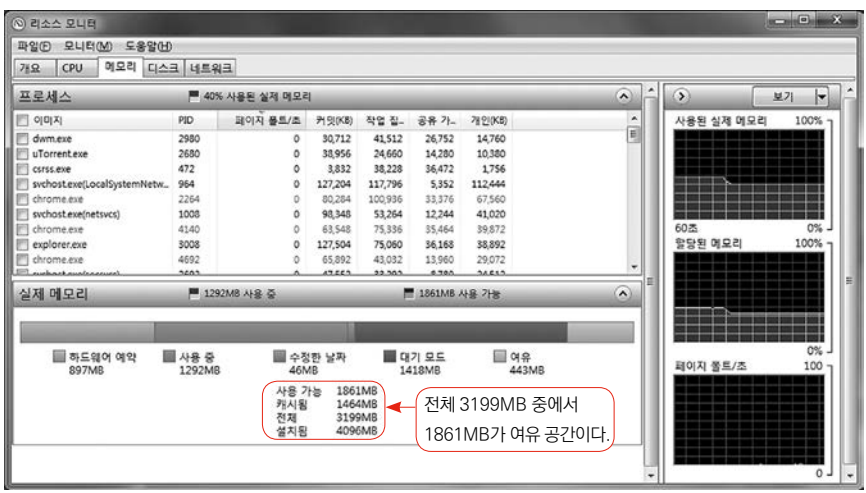
예를 들어, 컴퓨터가 32비트 CPU를 사용한다면 사용할 수 있는 전체 메모리 공간은 약 4GB가 된다. 하지만 컴퓨터에 설치된 메모리가 1GB밖에 안 된다면 1GB 이상의 프로그램이나 데이터는 사용할 수 없다. 이때 페이징을 사용하면 1GB는 실제 메모리에, 그리고 나머지 3GB는 디스크에 할당하여 사용한다. 실제 메모리나 디스크에 있는 가상 메모리는 페이지Page라는 단위로 관리되며 일반적으로 4KB가 기준 단위다.

CPU가 데이터를 처리하기 위해서는 데이터가 실제 메모리에 적재되어야 하는데 만약 처리해야 하는 데이터가 디스크에 저장되어 있으면 실제 메모리에 있는 페이

05 · 일반적으로 페이지 크기는 4KB다.

지 중에서 사용하지 않는 하나를 다시 디스크에 저장하고, 필요한 데이터 페이지를 디스크에서 가져와서 비워진 실제 메모리 공간에 적재하는 방식으로 동작한다. 동작하는 기본 원리는 캐시와 거의 같다고 볼 수 있다. 필요한 데이터가 캐시에 없으면 이를 '캐시 미스'라고 해서 메모리에서 데이터를 가져와서 캐시에 저장한다. 캐시 미스와 비슷하게 실제 메모리에 필요한 데이터가 없을 때는 '페이지 폴트Page Fault'라고 하며 페이지 폴트가 발생하면 실제 메모리에서 페이지 하나를 비우고 필요한 페이지를 디스크에서 가져와 메모리에 적재하는 과정을 거친다. 다음 그림은 개인 컴퓨터에서 사용하고 있는 메모리에 대한 현황이다.

[그림 1-6] 윈도우 작업 관리자를 통해 확인할 수 있는 시스템 자원량



실제 메모리의 용량(4GB)에 비해서 사용하는 프로그램이 많지 않아서 페이지 폴트가 거의 발생하지 않고 있다. 페이징 기법은 성능의 이점뿐 아니라 실제 물리적인 메모리보다 더 큰 메모리 공간을 사용할 수 있도록 해주므로 기본적으로는 사용하는 것이 좋다. 그리고 페이징 기법 때문에 고려해야 할 부분도 거의 없어서 이론적인 부분에 대한 이해는 많이 필요하지 않다.

그렇다면 왜 페이징을 이야기하는가? 앞에서도 설명했듯이 디스크 일부를 메모리처럼 사용할 수 있는 기법이 페이징이다. 이는 거꾸로 이야기해서 파일을 마치 메모리의 일부처럼 사용할 수도 있다는 뜻이 된다. 윈도우 운영체제는 CreateFileMapping과 MapViewOfFile 함수를 사용하고, 유닉스는 open과 mmap 함수를 이용해서 파일을 마치 메모리의 일부인 것처럼 사용할 수 있다.[06] 그렇다면 이렇게 파일을 메모리에 매핑할 때 어떤 이점이 있는지 알아보자.

예를 들어, 프로그램이 특정 파일을 열어서 100번 오프셋부터 20 크기만큼의 구조체 데이터를 10개 읽어와서 처리한 뒤 이를 다시 파일의 같은 위치에 써야 한다고 가정해 보자. 이를 기존의 파일 처리 함수를 사용하면 다음과 같이 작성할 수 있다.

[코드 1-4]

```
    int main()
    {
       int fd;
       int i;
       struct my_data data[10];
       fd = open("./data.dat", O_RDWR);
       // 100번째 오프셋으로 이동한 뒤 데이터를 읽어온다.
01     lseek(fd, 100, SEEK_SET);
       for(i=0; i<10; i++)
02        read(fd, &data[i], 20);
       ……

       // 읽어온 데이터를 처리한다.
       for(i=0; i<10; i++)
       {
          data[i].a = data[i].b + data[i].c;
```

06 각 함수의 사용 방법은 MSDN이나 man 명령으로 통해서 쉽게 알 수 있으므로 자세한 설명은 생략한다.

```
        data[i].b = data[i].b + data[i].c / 10;
        data[i].c = 0;
    }

    // 100번째 오프셋으로 이동한 뒤 데이터를 기록한다.
    lseek(fd, 100, SEEK_SET);
    for(i=0; i<10; i++)
        write(fd, &data[i], 20);

    return 0;
}
```

01 파일에 접근하는 위치를 파일 시작 위치에서 100바이트 뒤로 옮긴다.

02 파일의 지정된 위치에서 20바이트 만큼씩 읽어온다. 읽어올 때마다 파일 접근 위치는 읽은 만큼의 위치로 이동한다.

그리 복잡한 내용은 없다. 구성 자체가 짧으니 복잡할 것도 없지만, 이렇게 파일에서 값을 읽어오고 쓰는 것이 프로그램이 길어지면 상당히 귀찮고 골치 아픈 경우가 많다. 이럴 때 파일을 메모리에 매핑하면 어떻게 될까?

[코드 1-5]

```
int main()
{
    int fd;
    int i;
    struct stat statBuf;
    struct my_data *data;
    void *file_in_memory;
    fd = open("./data.dat", O_RDWR);

01  fstat(fd, &statBuf);
```

```
02      file_in_memory = mmap(NULL, statBuf.st_size, PROT_READ ¦ PROT_WRITE,
    MAP_SHARED, fd, 0);
03      data = (struct my_data *)(file_in_memory + 100);

        // 읽어온 데이터를 처리한다.
        for(i=0; i<10; i+ +)
        {
           data[i].a = data[i].b + data[i].c;
           data[i].b = data[i].b + data[i].c / 10;
           data[i].c = 0;
        }

        munmap(file_in_memory, statBuf.st_size);
        close(fd);

        return 0;
    }
```

02 마치 메모리의 일부처럼 파일 내용을 사용할 수 있도록 주소를 할당한다. 파일 시작 주소는 file_in_memory 변수에 저장된다.

03 배열에 접근하듯 파일에 접근한다. 이전과 같이 read를 사용하지 않는다.

준비 과정이 약간 복잡해졌지만, 파일에서 데이터를 읽어오거나 쓰는 과정은 사라졌다. [코드 1-4]는 유닉스 운영체제에서 사용할 수 있는 코드지만, 기본 원리가 같으므로 다른 함수를 사용하면 윈도우 운영체제에서도 충분히 사용할 수 있다. [코드 1-5]의 01 ~ 03이 파일을 메모리에 매핑하는 부분이다. mmap이라는 함수는 주어진 파일 지시자를 사용해서 해당 내용을 페이지에 할당하고 메모리의 어디에 해당 파일이 매핑되었는지 그 주소를 반환한다.

일단 매핑이 성공하면 반환된 주소를 사용하여 마치 배열을 사용하듯이 파일의 내용을 읽고 파일에 내용을 쓸 수도 있다. 앞에서 설명한 대로 파일의 100번째 오프

셋부터 사용할 구조체 데이터가 저장되어 있으므로 구조체 배열로 사용할 포인터 변수인 data의 주소값을 'mmap의 반환 주소 + 100'으로 지정한다. [코드 1-5]에서 `03` 부분이 이 내용이다. 이제 data 변수에 저장된 주소는 파일에서 100번째 오프셋부터 시작하는 구조체 데이터의 가장 처음을 가리키게 되고 data 변수를 배열처럼 사용하여 파일에 기록된 내용을 읽고 쓸 수 있다.

물론 페이지에 매핑된 영역을 더는 쓸 필요가 없다면, munmap 함수를 호출해서 해당 메모리 영역을 해제해야 한다. 운영체제가 페이지 폴트가 발생할 때 또는 주기적으로 내용이 바뀐 페이지를 디스크에 기록하므로 파일에 데이터를 기록하는 것은 신경 쓸 필요가 없다.

[코드 1-5]는 페이징 기법을 사용하는 간단한 예시지만, 페이지를 활용하는 방법은 실로 무궁무진하다. 메모리처럼 사용하므로 사용 방법이 매우 직관적이며 파일 내용에 대한 공유도 쉽다. 또한, 읽기 전용의 파일이나 내용일 때는 메모리 공간을 덜 차지하고 여러 프로세스가 동일한 내용을 공유할 수 있어서 공간 효율성을 극대화할 수 있다. 예를 들어, 운영체제는 읽기만 하는 공용 라이브러리 코드의 경우 메모리 공간에 한 번만 적재하고 해당 라이브러리 코드를 요청하는 모든 프로세스에 동일한 페이지 주소를 알려준다. 즉, 100개의 프로세스가 하나의 표준 라이브러리를 사용하며 표준 라이브러리 코드 크기가 4KB라면, 메모리에 적재되는 표준 라이브러리 코드의 전체 크기는 4KB × 100이 아니라 그냥 4KB가 된다.

1.3 표준 입출력

다시 처음으로 돌아가서 printf 구문을 생각해보자. printf 함수는 지정된 포맷이나 문자열을 사용해서 프로그래머가 원하는 문장을 화면에 출력하는 함수다. 프로그래밍에 입문하는 사람이라면 가장 처음 접하는 함수의 하나가 아닐까 하는데 이 printf 함수는 굉장히 단순하다.

printf 계열의 함수는 주어진 포맷Format에 따라서 주어진 인자를 적당히 가공한 다음 가공된 배열을 put과 같은 함수를 통해 표준 출력Standard Output으로 전송한다. 표준 입력Standard Input이나 출력은 컴퓨터에서 기본으로 제공하는 입출력 기능으로, 디바이스 드라이버가 없더라도 이들은 초기에 사용 가능한 상태로 제공된다. 이것이 가능한 이유는 컴퓨터에 장착된 BIOSBasic Input/Output System 때문이다.

메인보드에 장착되는 이 시스템은 ROM이나 플래시 메모리 형태로 제공되며 가장 낮은 수준에서 시스템의 입출력을 담당한다. 디바이스 드라이버를 메모리에 적재하고 실행하는 것은 운영체제가 처리하는 일이다. 하지만 운영체제가 이런 일을 처리하기 위해서는 컴퓨터에 설치된 각종 하드웨어를 사용할 수 있어야 하지만, 디바이스 드라이버가 없으면 정확한 제어가 불가능하다.

BIOS는 부팅과 동시에 가장 기본적인 입출력 장치를 설정하고 이 정보를 운영체제에 제공한다. BIOS나 운영체제에서 가장 기본적인 출력 장치는 모니터이며 가장 기본적인 입력 장치는 키보드다. printf 함수가 사용하는 출력은 기본으로 모니터이며 scanf와 같은 함수가 사용하는 표준 입력은 키보드다. 이 표준 입출력은 구현이나 의미, 사용법이 운영체제에 따라서 조금씩 다르나 크게 다르지 않으며 대부분의 운영체제가 제공한다.

표준 입출력[07]은 과거에 사용하던 터미널Terminal이라는 단말 장치에서 유래되었다고 볼 수 있다. 터미널은 오직 모니터와 키보드 그리고 이들을 제어하기 위한 작은 칩만 포함하고 있고 독자적으로 작업을 수행할 능력은 갖추지 않은 장치였다. 컴퓨터가 아주 비싸고 덩치만 컸던 시절에 사람들이 컴퓨터를 사용하기 위해서 줄을 서서 기다리다가 이것이 별로 좋지 않은 방법이라는 것을 깨닫고 컴퓨터에 접속하여 직접 작업을 처리할 수 있도록 터미널이라는 장치를 개발하고 보급하기 시작했다. 따라서 터미널은 모니터와 키보드만 있으면 충분한 장치였다.

07 표준 입출력을 일컬어 Standard stream이라고도 지칭한다.

서론이 좀 길었는데, 표준 입출력은 어디까지나 운영체제가 전담하고 처리하는 기능이다. 도스와 같은 비선점형 운영체제는 다르지만, 요즘 운영체제는 대부분이 선점형이기 때문에 어느 한 프로그램이 표준 입출력을 독식해서 점유하거나 표준 입출력을 하드웨어 수준으로 구현하고 사용하는 일은 드물다. 프로그램은 표준 출력으로 문자열을 전송하고, 키보드로 입력된 문자열을 표준 입력으로 읽어오면 그만이다.

하지만 운영체제에서 보면 표준 입출력은 디스크와 마찬가지로 관리해야 하는 대상이다. 운영체제는 디스크를 처리하는 것과 같이 표준 입출력을 처리하기 위한 버퍼를 따로 설정하여 관리한다. 프로그램이 문자열을 모니터에 출력해 달라고 요청한다고 해도 바로 화면에 출력하지 않고 이 버퍼에 쌓아두게 된다. 그리고 필요한 시점이 되면 화면을 갱신하고 버퍼에 쌓인 문자열을 전부 화면에 내보낸다. 이렇게 버퍼를 비우는 행위를 대개는 플러시Flush라고 한다. 플러시를 수행하는 함수는 언어 대부분에서 제공하는데 그 이유는 운영체제나 시스템이 사용하는 버퍼 때문이다.

문제는 이 버퍼의 위치에 따라서 예상하지 못한 입력이나 출력 결과가 나올 때가 있다는 점이다. 일반적으로 입출력 버퍼는 프로세스별로 가지고 있는 커널 메모리 영역에 존재한다. 프로세스 내부에 포함되므로 어떤 이유로 프로세스의 메모리가 그대로 복사된다면 버퍼에 있는 표준 입출력의 내용 또한 그대로 복사될 수 있다.

유닉스 운영체제는 프로세스를 만들기 위해서 fork() 시스템 콜과 exec() 시스템 콜을 사용한다. 이 중에서 fork() 시스템 콜은 현재 프로세스가 사용하는 메모리를 그대로 복사해서 사용한다. 더 엄밀히 이야기해서 'copy on write'라는 기법을 사용한다. 이는 두 프로세스가 같은 메모리를 사용하다가 어느 한 프로세스가 변수값을 변경하는 일이 생기면 그때야 메모리를 따로 복사하고 각 프로세스가 서로 다른 메모리를 사용한다.

이렇게 fork() 시스템 콜을 사용해서 프로세스를 복사하고 사용하는 것은 웹 서비

스를 제공하는 데몬과 같은 프로세스에서 주로 사용한다. 하나의 프로세스가 계속 사용자 요청을 기다리다가 요청이 들어오면 자신과 동일한 프로세스를 새로 만든 다음, 새로 만들어진 프로세스에서 사용자 요청을 처리하고 종료하는 방식으로 구성되는 경우가 많다. 프로세스가 복사되는 과정에서 커널 메모리 영역도 복사되는데 이때 플러시 되지 못한 버퍼 내용이 있으면 이것은 복사된 프로세스도 그대로 가지고 있어서 예기치 않은 출력 결과를 보일 수도 있게 된다. 다음 코드를 살펴보자.

[코드 1-6]

```
    #include <stdio.h>
    #include <unistd.h>

    int main()
    {
       pid_t processID;
       printf("Hello, World!");
01     processID = fork();
       return 0;
    }
```

01 이 시점에서 프로세스 하나가 새로 생성된다. 두 프로세스는 부모와 자식이며 메모리 상태는 프로세스의 아이디와 같은 고유한 부분을 제외하고 동일하다.

이 코드는 'Hello, World!'라는 문자열을 화면에 출력한 다음 fork() 시스템 콜을 호출하여 자신과 같은 프로세스를 하나 만들어내고 바로 종료한다. 생성된 프로세스는 생성되는 시점 다음의 코드를 실행하기 때문에 printf 구문을 실행하는 것이 아니라 바로 종료한다. 따라서 화면에는 한 줄의 'Hello, World!' 문자열만 출력되어야 정상이다. 하지만 정말 그럴까?

[실행 결과]

```
$ gcc fork.c
$ ./a.out
Hello, world! Hello, world! $
```

01 Hello, world

01 이 문자열은 프로세스가 만들어지기 전에 출력되어야 하나, 그러지 못했다.

우선 printf 구문에 사용된 문자열에 개행 문자(\n)가 없다는 점에 유의하자. 개행 문자가 없으므로 문자열 바로 다음에 프롬프트 문자($)가 출력된다. 이 개행 문자는 단순한 줄 바꿈 외에도 운영체제에 따라 다양한 의미를 가지는데 유닉스는 터미널의 설정에 따라 조금씩 다르지만, 출력 버퍼를 플러시하라는 의미도 포함한다.

> **NOTE**
>
> 생성되는 자식 프로세스는 부모와 동일한 메모리 구조와 값을 가진다. 입출력 버퍼가 비워지지 않은 상태에서 프로세스가 생성되고 종료하면 두 프로세스는 모두 동일한 메시지를 출력하게 되므로 사용자 입장에서는 이들을 구분하기가 힘들다.

개행 문자가 입력되지 않았기 때문에 운영체제는 커널에 있는 버퍼의 내용을 즉시 출력하지 않고, 다음 개행 문자가 입력될 때까지 계속 입력을 기다린다. 이 상태에서 fork() 시스템 콜에 의해 프로세스의 메모리가 그대로 복사되고 두 개의 프로세스가 종료하면 두 프로세스는 모두 커널 메모리 영역에 'Hello, World!'라는 문자열을 출력하지 못한 상태로 기다리다가 프로세스가 종료하기 직전에 이 버퍼를 모두 비운다. 따라서 예상하지도 않게 문자열이 두 번이나 출력되는 것이다. 이는 매우 간단한 방법으로 해결할 수 있다. 개행 문자를 쓰거나 표준 출력 버퍼를 강제로 비우면 된다.

[코드 1-7]

```
#include <stdio.h>
#include <unistd.h>

int main()
{
   pid_t processID;
   printf("Hello, World!");
01 fflush(NULL);
   processID = fork();

   return 0;
}
```

01 입출력 버퍼를 강제로 비우도록 '시도'한다.

fflush( ) 함수는 인자로 NULL이 주어지면 모든 출력 버퍼를 비운다. 이렇게 하면 fork( ) 시스템 콜이 호출되기 전에 커널 메모리 영역의 출력 버퍼가 모두 비워지고, 결과적으로 문자열은 한 번만 출력된다.

[실행 결과]

```
$ gcc fork.c
$ ./a.out
Hello, world! $
```

여전히 개행 문자가 포함되어 있지 않아서 문자열과 프롬프트가 나란히 출력되지만, 여기서 중요한 것은 어느 시점에 출력 버퍼가 비워지느냐는 점이다. 윈도우 운영체제는 fork()와 같은 시스템 콜을 쓰는 일이 흔하지는 않지만, 프로세스의 메모리를 복사한다는 점은 유사하므로 프로세스를 복사해서 사용할 때 이 점을 유의해야 한다.