The Evolution of the Unix Time-sharing System

수학과 김호진 (2016314786)

ABSTRACT

본 논문은 유닉스(Unix) 운영체제의 초기 개발 역사를 간략히 설명한다. 그 중에서도 파일 시스템의 진화, 프로세스 제어 메커니즘, 파이프라인 명령어의 개념에 초점을 맞추며, 시스템 개발 과정에서의 사회적 여건에도 어느 정도 관심을 둔다.

Introduction

지난 몇 년간 유닉스 운영체제는 매우 광범위하게 사용되어 왔다. 유닉스 운영체제의 주요특징들은 많은 사람들에게 알려져 있다. 1974년 기록된 최초의 등장 이후, 많은 수정이 있었지만 근본적인 변화는 거의 없었다. 그러나 유닉스는 1974년이 아닌 1969년에 탄생했으며, 유닉스의 개발에 대해서는 알려진 바가 없다. 본 논문에서는 시스템의 진화과정에 대한 기술적, 사회적 역사를 기술한다.

Origins

Bell Laboratories의 컴퓨터 공학에 있어, 1968년부터 1969년까지의 기간은 Multics 프로젝트의 철수가 불가피하게 되면서 다소 불안정한 시기였다. Labs computing community의 전반적인 문제는 Multics가 어떤 종류의 사용 가능한 시스템도 제공하지 못한다는 점에 있었다. 대부분의 시간 동안 Murray Hill 컴퓨터 센터는 GE 635를 부적절하게 시뮬레이션 하는 값비싼 GE 645 기계를 운영하고 있었으며 이 기간 동안 컴퓨팅 서비스와 컴퓨팅 연구가 조직적으로 분리되었다.

Multics의 쇠퇴와 몰락은 유닉스의 시작에 직접적인 영향을 미쳤다. 당시 Multics는 많은 사용자들을 지원할 수 없었음에도 불구하고, 터무니없는 비용이 들었다. 그럼에도 프로그래밍을 할 수 있는 좋은 환경을 넘어 fellowship까지 형성할 수 있는 시스템을 보존하고자 했기 때문에 1969년 Multics의 대안을 찾기 시작했다. 운영체제를 작성하기로 계획된 중간 규모의 기계를 구매하기 위해서 DEC PDP-10과 SDS Sigma 7에 대한 제안을 했다. 하지만 많은 돈이 드는 것에 비해 계획은 모호했으며, 그 당시 운영체제는 경영진에게 있어 업무를 지원할 만큼 매력적인 분야가 아니었기에 제안은 거절당했다.

같은 해 Thompson, R. H. Canaday, Ritchie는 칠판과 노트에 파일 시스템의 기본 디자인을 개발하였고 이는 추후 유닉스의 핵심이 되었다. 이 기간 동안 Thompson은 여러 형태로 운영체제를 만들고자 했다. 또한 그는 제안된 파일 시스템 설계의 성능과 프로그램의 페이징 동작에 대한 꽤 상세한 시뮬레이션을 Multics에 작성했다. 더 나아가 GE-645를 위한 새로운 운영체제를 만들기 시작했고, 기계에 인사말을 타이핑하는 것이 가능한 초보적인 운영체제 커널을 만들기까지 하였다.

또한 Thompson은 'Space Travel'이라는 게임을 개발했다. 이는 Multics에서 최초로 작성된 이후 GECOS용 Fortran (GE용 운영 체제, 추후 Honeywell, 635)으로 옮겨졌다. GECOS 버전에서 게임 상태의 디스플레이는 명령을 입력해야 했기 때문에 조작하기 어려웠고, 큰 컴퓨터에서 CPU 시간을 위해 75달러 정도의 비용이 든다는 단점이 있었다. 다행히 뛰어난 디스플레이 프로세서를 갖춘 PDP-7 컴퓨터를 찾는 데는 그리 오랜 시간이 걸리지 않았다. 다만, 기존의 모든 소프트웨어와 맞지 않았기 때문에 이 기계를 작동시키기 위해 부동소수점 연산 패키지, 디스플레이를 위한 그래픽 문자의 포인트별 사양, 화면 한 구석에 입력된 위치의 내용을 지속적으로 표시하는 디버깅 서브시스템 등을 작성해야만 했다. 이 모든 것은 GECOS에서 작동하는 cross-assembler를 위한 assembly language로 작성되었고 종이 테이프로 생산되어 PDP-7에 제공되었다.

Thompson은 이전에 종이에 디자인했던 파일 시스템을 구현하기 시작했다. 실행방법이 없는 파일 시스템은 쓸모 없는 제안에 불과했기 때문에 프로세스의 개념을 담은 작동 중인 운영체제의 요구 사항을 구체화했다. 곧이어 파일을 복사, 인쇄, 삭제 및 편집하는 수단과 간단한 명령 인터프리터(쉘)와 같은 소규모 사용자 수준의 유틸리티까지 등장했다. 지금까지의 모든 프로그램들은 GECOS를 사용하여 작성되었으며 파일들은 종이 테이프를 통해 PDP-7로 전송되었다. 1970년이 되어서야 Brain Kernighan이 '유닉스'라는 이름을 제안했지만, 이미 'Multics'에 대한 무모한 장난으로부터 오늘날 우리가 알고 있는 운영체제가 탄생했다.

The PDP-7 Unix file system

구조적으로 PDP-7 유닉스의 파일 시스템은 오늘날과 거의 동일하며 다음을 가진다.

1) i-list: 각각의 파일을 설명하는 i-nodes 선형 배열. 지금보다 적은 i-node가 포함되어 있지만 파일의 보호 모드, 파일의 유형 및 크기, 콘텐츠를 보관하는 physical blocks의 목록 등의 필수 정보는 동일하게 가지고 있다.

2) Directories: 이름 및 관련된 i-number를 포함하는 특수 파일.

3) 장치를 설명하는 특수 파일: 장치 사양은 i-node에 명시적으로 포함되어 있지 않은 대신 숫자로 암호화되었다: 특정 i-number는 특정 파일에 대응된다.

주요 file system call 역시 처음부터 존재했다. 아래 논의된 예외를 제외한 Read, write, open, create(sic), close는 현재와 유사했다. 사소한 차이는 PDP-7이 word-addressed machine이기 때문에 입출력 단위가 byte가 아닌 word라는 점이었다. 또 다른 차이점은 터미널에 대한 erase 및 kill processing이 부족하다는 것이었다. 사실상 터미널은 항상 원시(raw) 모드였기 때문에 일부 프로그램(특히 shell과 editor)은 erase-kill processing을 구현하는 데 있어 어려움을 겪었다.

PDP-7 파일 시스템은 오늘날 파일 시스템과 상당히 유사하면서도, 한 가지 면에서 매우 달랐다. 경로 이름이 없었으며 시스템에 대한 각 파일 이름 인수는 현재 디렉터리에 대해 사용된 단순한 이름이었다. 그 대신 일반적인 유닉스 관점의 link가 존재했다. 경로 이름이 부족해지면서 link는 주요한 수단이 되었다.

link 호출(call)은 'link(dir, file, newname)' 형식으로 이뤄지며, dir은 현재 디렉터리에 있는 디렉터리 파일을, file은 그 디렉토리 내에 존재하는 entry를, newname은 기존 디렉토리에 더해질 link의 이름을 의미한다. dir가 현재 디렉토리에 있어야 하므로 오늘날 디렉토리로의 link에 대한 금지가 시행되지 않았음을 알 수 있다.

사용자가 모든 관심 디렉토리에 대한 link를 유지할 필요가 없도록 각 사용자의 디렉토리에 대한 entries를 포함하는 디렉토리 dd가 존재하기는 했지만, 사용하기 어렵게 만들어졌고 시스템이 실행되는 동안 디렉터리를 만들 방법이 없다는 문제가 있었다. dd convention은 chdir 명령어를 비교적 편리하게 만들었으며, 여러 인수들을 사용하면서 현재 디렉터리를 각 명명된 디렉터리로 차례로 전환했다. 그러면서 'chdir dd ken'을 통해 디렉터리 ken으로의 이동을 가능하게 만들었다.

파일 시스템의 구현에서 가장 큰 불편함은 경로 이름의 부족보다 configuration 변경의 어려움이었다. 디렉토리와 특수 파일은 모두 디스크가 재생성 될 때에만 만들어지는데 장치에 대한 코드가 시스템 전체에 널리 퍼져 있었기 때문에 새로운 장치를 설치하는 것은 매우 어려운 일이었다.

Multi-programming이 되지 않았기 때문에 파일 시스템을 구현한 운영체제 코드는 현재보다 획기적으로 단순화된 버전이었다. 오늘날 버퍼링 메커니즘의 전구체가 존재하기는 했지만 계산상 디스크 I/O와 겹치지 않았다. 그런데 기계에 index registers가 없었기 때문에 indirection이 자주 나타나게 되었고, instruction 실행 중 DMA 컨트롤러가 메모리에 접근할 수 없게 되었다. 그러면서 디스크가 전송하는 동안 간접적으로 주소가 지정된 명령어가 실행되면 오버런 에러가 발생했다. 이로 인해 제어권이 사용자에게 반환될 수 없었으며, 디스크가 실행 중인 상태에서 일반 시스템 코드를 실행할 수 없었다. 또한 clock와 terminal의 interrupt routine이 항상 실행되도록 해야 했기 때문에 indirection을 피하기 위해서 매우 이상한 방식으로 코드화가 되었다.

Process control

'프로세스 제어'란 프로세스가 생성되고 사용되는데 쓰이는 메커니즘을 의미한다. 오늘날 시스템은 fork, exec, wait 및 exit를 호출하여 이러한 메커니즘을 구현한다. 초기부터 현재의 형태와 거의 동일하게 존재했던 파일 시스템과는 다르게 프로세스 제어 체계는 PDP-7 유닉스에서 사용된 이후 상당한 변화를 겪었다.

오늘날 shell에 의해 command가 실행되는 방식은 다음과 같이 요약할 수 있다.

1) 셸이 터미널에서 command line을 읽는다.
2) fork를 통해 child process를 만든다.
3) child process는 exec을 사용하여 파일에서 command를 호출한다.

4) parent shell은 wait를 사용하여 child(command) process가 exit를 호출해 종료될 때까지 기다린다.
5) parent shell이 1단계로 돌아간다.

프로세스(독립적으로 실행되는 entity)는 PDP-7 유닉스 초기부터 존재했다. 다만 fork, wait, exec는 존재하지 않았으며 exit가 존재하기는 했지만, 그 의미가 다소 달랐다. shell의 main loop는 다음과 같다.

1) 셸이 열려 있는 모든 파일을 닫은 다음 표준 입력 및 출력을 위한 터미널 특수 파일을 연다.
2) 터미널로부터 command line을 읽는다.
3) command를 지정하는 파일에 link하고 파일을 연 다음 link를 제거한다. 그리고 부트스트랩 프로그램을 메모리의 맨 위로 복사한 뒤 그 곳으로 점프한다. 부트스트랩 프로그램은 셸 코드를 통해 파일에서 읽은 다음 command의 첫 번째 위치로 점프한다(exec).
4) command는 작업을 수행한 후 exit 호출을 통해 종료된다. exit 호출은 시스템이 terminated command을 통해 셸의 새로운 사본을 읽은 뒤 시작 부분으로 점프하도록 만든다(step 1로 보낸다).

pipes와 filters는 물론, 백그라운드 프로세스나 셸 명령 파일도 지원하지 않았지만 IO redirection('<' 및 '>')이 도입되었다. Redirection의 구현은 셸 위의 표준 입력이나 출력을 적절한 파일로 교체하기만 하면 되는 매우 간단한 작업이었다. 후속 개발을 위해 중요한 점은 셸을 운영체제의 일부가 아닌 파일에 저장된 사용자 수준의 프로그램으로 구현하는 일이었다.

터미널당 하나의 프로세스가 있는 프로세스 제어 체계의 구조는 CTSS, Multics, Honeywell TSS, IBM TSS 및 TSO와 같은 많은 interactive system 구조와 유사했다. 일반적으로 이러한 시스템은 detached computation이나 command file과 같은 유용한 기능들을 구현하기 위해 특별한 메커니즘을 필요로 했는데 이것이 몇 가지 문제점을 유발했다. 셸은 command마다 새로 실행되었기 때문에 /dev 디렉터리가 없었으며, command 간에 메모리를 보유하지 못했다. 이러한 문제점을 해결하기 위해서 추가적인 파일 시스템 규칙이 필요하게 되었다.

현대적인 형태의 프로세스 제어는 빠르게 설계되고 구현되었다. 설계의 특징들 중 일부는 이미 존재하는 것을 기반으로 변화된 코드였기 때문에 쉽게 이해할 수 있었으며 기존의 시스템과 잘 들어맞았다. fork와 exec 함수의 분리는 이를 설명하는 좋은 예시다. 유닉스에서 forked process는 명시적인 exec를 수행할 때까지 부모 프로그램과 동일한 프로그램을 계속 실행하며, fork의 초기 구현에는 다음의 항목들을 필요로 했다.

1) 프로세스 테이블의 확장
2) 이미 존재하는 swap IO 기본 요소를 사용하여 현재 프로세스를 disk swap 영역으로 복사하고 프로세스 테이블을 일부 조정하는 fork call 추가

combined fork-exec의 경우, exec가 존재하지 않았기 때문에 훨씬 더 복잡했다. exit system call은 프로세스가 프로세스 테이블 항목만을 정리하고 제어는 포기하게 되면서 상당히 단순화되었다. 흥미롭게도, wait이 된 주요 요소들은 현재의 체계보다 일반적이었다. 주요 요소 쌍이 명명된 프로세스 간에서 one-word message를 보내는 과정은 'smes(pid, message)', '(pid, message) = rmes()' 형태로 이뤄진다. 이때 parent shell은 command를 실행하기 위한 프로세스를 만든 후 smes를 통해 새로운 프로세스에 메시지를 보낸다. command가 종료되면, shell에서 차단된 smes call은 target process가 존재하지 않는다는 오류 표시를 반환하게 되고 shell의 smes은 wait와 같은 역할을 하게 된다.

초기화 프로그램과 각 터미널의 셸 사이에서는 메시지가 제공하는 일반성을 더 많이 이용하는 프로토콜이 사용되었다. 초기화 프로세스는 각 터미널에 대한 셸을 생성한 다음 rmes를 발행했다. 각 셸은 입력 파일의 끝을 읽을 때 기존의 'I am terminating'라는 메시지를 초기화 프로세스에 전송하기 위해 smes을 사용했으며, 이는 해당 터미널에 대한 새로운 셸 프로세스를 재생성 했다.

새로운 프로세스 제어 체계는 detached process('&' 포함)나 셸을 명령으로 재귀적으로 사용하는 것과 같이 구현하는데 있어 매우 중요한 기능들을 즉각적으로 만들어냈다. 대부분의 시스템은 'batch job submission' 기능을 비롯하여 상호적으로 사용되는 파일과 구별되는 특수 명령 인터프리터를 제공해야 했다.

그러던 와중, chdir(현재 디렉터리 변경) 명령이 작동을 멈춘 것이 발견되었다. fork의 추가가 어떻게 chdir call을 망가뜨렸는지 알아내기 위해서 코드를 읽고 점검을 하였다. 이전 시스템에서 chdir은 터미널에 존재하는 프로세스의 current directory를 변경하는 일반적인 명령이었다. 그러나 새로운 시스템에서 chdir는 해당 프로세스를 실행하기 위해 생성된 프로세스의 현재 디렉토리를 올바르게 변경하기는 하지만, 이 프로세스가 즉시 종료되면서 parent shell에 아무런 영향을 미

치지 못하게 되었다. 이를 해결하기 위해서 chdir을 셸 내부에서 실행되는 특별한 command로 만들어야 했다.

시간이 지나 시스템과 새로운 프로세스 제어 체계 사이의 또 다른 불일치가 발견됐다. 기존의 경우, 열려 있는 각 파일과 관련된 읽기/쓰기 포인터는 파일을 여는 프로세스 내에 저장되었다. 이러한 구조의 문제는 command file을 사용하려고 했을 때 분명해졌다. 단순 명령 파일에 'ls', 'who'라는 command가 차례로 포함되어 있다고 가정하자. 그리고나서 'sh comfile > output'가 실행되면 다음의 일들이 순차적으로 발생하게 되었다.

1) 메인 셸이 새로운 프로세스를 생성하고, outfile을 열어 표준 출력을 수신한 뒤 셸을 재귀적으로 실행한다.
2) 새로운 셸은 ls를 실행하기 위한 또 다른 프로세스를 생성하며, 이는 output 파일에 기록을 마치고 종료된다.
3) 다음 command를 실행하기 위한 다른 프로세스가 생성된다. 그러나 출력을 위한 IO 포인터는 셸의 포인터로부터 복사되고, 값은 여전히 0이다. 왜냐하면 셸이 출력에 기록된 적이 없고 IO 포인터는 프로세스와 연관되어 있기 때문이다. 그러면서 who의 output은 선행 ls command의 output을 덮어쓰고 파괴하게 한다.

이 문제를 해결하기 위해서는 open file의 IO 포인터를 포함하는 새로운 시스템 테이블을 만들어야만 했다.

IO Redirection

'>'와 '<' 문자를 사용한 IO redirection을 위한 표기법은 비교적 일찍 나타났으며 Multics의 아이디어로부터 영감을 받았다. Multics는 다양한 장치, 파일 또는 특수 스트림 처리 모듈을 통해 동적으로 redirected될 수 있는 IO 스트림을 구현하는 일반적인 IO redirection 메커니즘을 가지고 있었다. 다만, Multics 프로젝트가 워낙 방대했기 때문에 Multics shell을 통합하는 것은 매우 간단한 일이었음에도 이전까지는 고려조차 되지 않았다. 마침내 Unix IO 시스템과 셸이 Thompson의 단독적인 통제 하에 있게 되면서 IO redirection이 구현되었다.

The advent of the PDP-11

1970년 초, PDP-7이 구식이 되면서 digital을 도입한 PDP-11의 인수를 제안하게 되었다. 이전에 요구했던 것보다 훨씬 적은 금액(약 65,000달러)이었으며, 단순히 불특정한 운영체제를 작성하는 것을 넘어 오늘날 'word-processing system'이라고 불리는 텍스트 편집과 서식을 위해 설계된 시스템을 만들고자 했다. 프로세서가 여름 말에 도착했지만 PDP-11가 최신 제품이었기 때문에 12월까지 사용가능한 디스크가 존재하지 않았다. 그래서 그 전까지는 PDP-7의 cross-assembler를 사용하여 core-only 버전의 유닉스가 작성되었다.

The first PDP-11 system

디스크가 도착한 뒤 시스템은 빠르게 완성되었다. Multi-programming이 없었기 때문에 PDP-11용 유닉스의 첫 번째 버전은 내부적인 구조면에서 PDP-7 시스템에 비해 상대적으로 미미한 발전을 이뤄냈지만, 사용자 인터페이스에서는 중요한 변화가 있었다. 완전한 경로 이름을 가진 디렉토리 구조는 현대적인 형태의 exec과 wait, 그리고 터미널을 위한 character-erase 및 line-kill processing과 같은 편의성을 지니게 되었다. 기업의 입장에서는 작은 크기가 가장 흥미로웠을 것이다.

PDP-7 유닉스의 유용성 증가는 PDP-11 유닉스가 특별한 목적의 시스템을 작성하는 데 사용될 수 있도록 만들었다. 그로 인해 McIlroy의 BCPL 버전에서 번역된 Multics의 PDP-7 버전으로부터 roff text formatter를 PDP-11 어셈블러 언어로 변환하게 되었다. 이것은 확장된 type-box로 필요한 대부분의 수학 기호를 인쇄할 수 있는 Teletype's model 37 terminal을 지원했으며, 'roff'는 줄 번호 페이지를 만들 수 있는 능력을 가지고 있었다.

메모리 보호가 없었고 단일 .5 MB 디스크가 없는 컴퓨터에서 새로운 프로그램의 테스트는 시스템을 쉽게 손상시킬 수 있었기 때문에 주의와 대담성을 필요로 했다. 또한 매우 작은 디스크 때문에 typists가 몇 시간 동안 작업을 할 때마다 더 많은 정보를 DEC테이프로 내보내야만 했다. 실험은 어려웠지만 결과는 성공적이었고 특허 부서는 유닉스를 채택했다. 이를 통해 PDP 11/45 시스템 중 하나를 만들 수 있도록 경영진을 설득할 수 있는 충분한 신뢰를 얻게 되었다.

Pipes

명령 파이프라인에서 사용되는 pipes는 운영체제와 command에 대한 유닉스의 엄청난 기여 중 하나다. Dartmouth Time-Sharing System의 '통신 파일'은 유닉스의 pipes가 하는 일과 거의 비슷한 역할을 하긴 했지만, 완전히 활용되지는 못했다.

Pipes는 1972년 PDP-11 버전의 시스템이 가동된 이후 coroutine을 특징짓는 비계층적 제어 흐름을 오랫동안 지켜본 M. D. McIlroy의 제안으로 유닉스에 등장하게 되었다. pipes가 구현되기 몇 년 전부터 그는 왼쪽과 오른쪽 피연산자가 각각 입력 및 출력 파일을 지정하는 이진 연산자로 간주되어야 한다고 제안했다. 그의 아이디어는 흥미를 끌었지만 즉각적인 행동에 불을 붙이지는 못했다. Infix 표기법은 급진적으로 보였고, 명령 매개변수를 입력 또는 출력 파일과 구별하는 방법을 알 수 없었다. 또한, command execution의 one-input one-output 모델은 너무나도 엄격해 보였다. 다행히도 McIlroy의 노력으로 운영체제에 pipes가 설치되었고, 새로운 표기법 역시 도입되었다. 이때 I/O redirection과 동일한 문자가 사용되면서 파일 혹은 출력의 redirection을 지정하거나, 이전 명령의 출력을 입력으로 지시하는 명령어 모두 '>' 뒤에 올 수 있게 되었다.

새로운 장비는 열광적인 반응을 얻었고, 곧 'filter'라는 용어가 만들어졌다. 많은 명령어들은 파이프라인에서 사용될 수 있도록 변경되었다. 그러나 얼마 지나지 않아 표기법의 몇 가지 문제점이 발견되었다. 가장 성가신 문제는 어휘 문제였으며, 일반성을 부여하기 위해 파이프 표기법이 '<'를 '>'에 대응하는 방식으로 input redirection을 받아들이면서 기호가 고유하지 않게 되었다. 결국 파이프라인의 구성요소를 분리하기 위해 고유한 연산자를 사용하는 현재의 표기법이 등장했다

IO redirection에서 Multics가 처리 모듈을 통해 소스 또는 싱크 역할을 하는 장치/파일로 이동하는 과정에서 IO 스트림을 지시할 수 있는 메커니즘을 제공한다고 했기 때문에 Multics의 stream-splicing은 유닉스 pipes의 직접적인 선구자처럼 보인다. 하지만 coroutines은 이미 잘 알려져 있었을 뿐만 아니라 Multics spliceable IO module의 구현은 모듈을 다른 용도로 사용할 수 있는 방식으로 특별히 코딩해야 했기 때문에 이를 사실이라고 보기에는 어렵다. 아마 진정한 유닉스 파이프라인의 특징은 simplex 방식으로 지속적으로 사용되는 동일한 명령어로 구성된다는 점일 것이다.

High-level languages

기존 PDP-7 유닉스 시스템의 모든 프로그램은 매크로가 없는 순수 어셈블리어로 작성되었다. 게다가, loader와 link-editor가 없었기 때문에 모든 프로그램은 그 자체로 완벽해야 했다. McIlroy가 구현한 McClure의 TMG가 최초로 보급된 직후, Thompson은 실제 컴퓨팅 서비스를 제공하기 위해 TMG에 Fortran을 작성하고자 했다. 그리고 그는 Fortran 대신 새로운 언어인 B의 정의와 컴파일러를 만들어냈다. B는 BCPL 언어의 영향을 많이 받았으며 컴파일러는 간단한 해석 코드를 제공했다. regular system call에 대한 인터페이스가 가능해지면서 시스템 프로그램(컴파일러, 어셈블리 등) 역시 작성되었다. PDP-11용 PDP-7B 크로스 컴파일러는 B로 작성되었으며, PDP-7 자체를 위한 B 컴파일러 또한 TMG에서 B로 변환되었다.

디스크가 도착하기 전 PDP-11에서 실행되었던 다중 정밀도 '데스크 계산기' 프로그램 dc가 있었지만 B는 이를 가져가지 않았다. 오직 어셈블러가 아닌 B로 운영체제를 다시 작성하는 것 만을 고려했으며 대부분의 유틸리티에 대해서도 마찬가지였다. 심지어 어셈블러도 어셈블러로 다시 작성되었다. 이러한 접근은 주로 interpretive code의 느림으로 인해 발생했다. 단어 지향 B 언어와 PDP-11의 byte-addressed 간의 불일치는 작지만 중요한 문제가 되었다.

그래서, C 언어가 되기 위한 작업이 1971년부터 시작되었다. 가장 중요한 분수령은 운영체제 커널이 C로 작성된 1973년에 일어났다. 이 시점에서 시스템은 multi-programming의 도입 등으로 인해 현대적인 형태를 가지게 되었다. 외부적으로 눈에 보이는 변화는 거의 없었지만, 체계의 내부 구조는 훨씬 더 합리적이고 일반적으로 변했다. 그리고 이는 C가 시스템 프로그래밍을 위한 보편적인 도구로써 유용하다는 것을 확신시켰다. 오늘날 어셈블러로 작성된 유일한 유닉스 프로그램은 어셈블러 그 자체만 존재하며, 대부분의 응용프로그램을 비롯한 사실상의 모든 유틸리티 프로그램들은 C 위에서 작성되었다. 그렇기 때문에 유닉스의 성공이 high-level language로부터 표현된 소프트웨어의 가독성, 수정성, 이식성으로부터 비롯된다는 것은 부정할 수 없어 보인다.

Conclusion

초기 버전의 유닉스가 제공하는 프로그래밍 환경은 지금 보면 매우 가혹하고 원시적인 것처럼 보인다. 하지만 그 당시에는 그렇게 보이지 않았다. 그러므로 10년 뒤에도 연속성을 가지고 기술의 진보를 돌아보기를 바란다.

My opinion

이번 학기 운영체제 강의를 수강하게 되었다. 컴퓨터 공학에서 운영체제가 상당히 중요한 부분을 차지한다는 것은 알았지만 본 논문에서 언급된 것처럼 주요 특징 정도만 아는 정도였고 개발과정에 대해서는 아는 바가 없었다. 이번 1주차 과제를 통해 평소 접하기 힘들었던 유닉스의 역사에 대해 알아갈 수 있어 좋은 경험이 되었다.

비록 현재는 유닉스를 쓰는 곳이 극히 적기는 하지만, 유닉스는 현대적 컴퓨터 운영체제의 원형이라고 할 수 있다. 리눅스, 윈도우와 더불어 유닉스는 오늘날 운영체제의 한 부분을 이루고 있으며 macOS, iOS의 뿌리가 되었다. 또한, 컴퓨터 운영체제 역사에서 중요한 운영체제이기도 하다. 오늘날 우리가 python과 더불어 흔히 첫번째 프로그래밍 언어로 접하게 되는 C는 이러한 유닉스 운영체제의 발전에 있어 중요한 역할을 하였다.

본 논문에서 언급된 바처럼 유닉스의 대부분은 high-level language인 C로 쓰여 졌다. C의 경우, 유닉스 운영체제 시스템 프로그래밍을 목적으로 Bell Laboratories에 의해 설계 개발된 프로그래밍 언어였기 때문에 매우 효율적이고 이식이 쉬운 언어였다. 또한 소스코드를 쉽게 구할 수 있기 때문에 다른 컴퓨터 하드웨어나 새로운 기종에 쉽게 이식할 수 있다는 장점을 가졌다. 이러한 장점에 더불어 유닉스가 프로그램을 실행할 수 있는 최상의 환경을 제공하였기 때문에 각종 편리한 프로그램 도구들이 발달할 수 있었으며 개발하기에 가장 적합한 환경으로 발전하였다. 그러면서 C언어는 자연스럽게 시스템 프로그래밍 언어의 표준이 되었다.

나를 비롯한 컴퓨터 공학을 전공하는 대다수의 학우들은 1학년 과목인 공학 컴퓨터 프로그래밍 또는 프로그래밍 기초와 실습 과목을 통해 C언어를 접해봤을 것이다. 하지만, 해당 과목에서 C언어의 특징이나 개발하게 된 역사에 대해서는 다루지 않거나 가볍게 짚고 넘어가기 때문에 대부분이 이에 대해 알지 못할 것이라 생각한다. 나 또한 지금까지 C를 개발하게 된 역사에 대해서는 크게 관심을 가지지 않았다. 하지만 본 논문을 읽어보면서 유닉스의 성공을 가져온 high-level language C가 어떻게 등장하게 되었는지 궁금증이 생겼고 이를 찾아보면서 C의 등장 배경과 해당 언어가 가지는 특징까지도 알게 되었다. 그리고 이를 기반으로 C가 가지는 구조적 특징을 이해하는데 도움을 받았다.

또한 유닉스에 대해 추가적으로 알아보면서 컴퓨터 네트워크 분야에도 유닉스가 큰 영향을 미쳤다는 사실을 알게 되었다. 유닉스가 TCP/IP 등의 네트워크 기능을 일찍부터 가지고 있었기 때문에 이더넷 같은 인터넷 기능을 사용하기 위해서는 유닉스를 사용해야만 했다. 또한 BSD 유닉스에서 운영체제가 인터넷에 접근하는 표준 인터페이스인 소켓이 만들어졌으며 일반적으로 소켓은 BSD 소켓을 일컫는 말이 되었다. 이처럼 유닉스는 컴퓨터 네트워크를 크게 발전시켰으며 오늘날 인터넷의 성공에도 지대한 영향을 주었다.

이번 과제를 하면서 가장 공감이 되었던 점은 Conclusion에서 정리된 저자의 말이었다. 유닉스 운영체제의 역사를 지켜보면서 단순한 프로그래밍 환경을 제공하는 운영체제에서 시작하여 짧은 시간동안 엄청난 발전이 이뤄졌음을 알 수 있었다. 하지만 아무리 단순해 보이는 기술들도 그 당시에는 부족한 점을 해결해주는 획기적인 발전이었다. 마찬가지로 최근에 개발된 수많은 기술들 역시 우리가 보기에는 엄청난 것처럼 보이며 더 이상 개선시킬 부분이 없어 보이지만 분명 그 기술들 속에도 불편하고 부족한 점은 존재할 것이며 미래에는 그 부분을 보완하고 개선한 또 다른 기술이 등장할 것이다. 그리고 그러한 미래에서 보는 오늘날의 기술들 또한 허점 많고 보완이 필요한 단순한 기술처럼 보일지도 모른다.

이처럼 컴퓨터 공학의 대부분의 발전은 부족함을 보완하는 과정 속에서 이루어졌다. 현대적 운영체제의 지대한 영향을 미쳤으며 찬란했던 과거를 가진 유닉스 역시 비용적인 측면에서 리눅스에게 경쟁력을 가지지 못하고, x86 프로세서 기반의 윈도우와의 완전한 이식이 이루어지지 않으면서 오늘날에는 쓰이는 곳이 크게 줄어들게 되었다. 하지만 Multics가 없었다면 유닉스도 없었던 것처럼, 유닉스는 리눅스를 비롯하여 BSD, 스마트폰 운영체제 등의 등장에 지대한 영향을 미쳤다. 또한 유닉스로부터 등장한 리눅스도 미래에 또 다른 운영체제가 등장한다면 엄청난 영향을 줄 것이 분명하다.

그렇기 때문에 우리는 10년 뒤에도, 그리고 그 이후에도 기술의 진보를 계속해서 돌아봐야 한다. 앞으로 운영체제 분야에서 기술의 진보가 언제, 얼마나 일어나게 될 지는 아무도 예상할 수 없지만, 기존의 기술이 필요로 하는 보완점을 잘 살펴본다면 그 방향이 어디로 갈 지는 알 수 있을 것이라 생각한다. 본 논문을 읽으면서 유닉스의 진화과정을 상세하게 알 수 있어서 정말 유익한 시간이었고 앞으로 운영체제의 역사가 어떤 방향으로 진행될지 새로운 기대를 가지게 되었다.