The Evolution of the Unix Time-sharing System

Summary

Abstraction & Introduction & Origins

이 논문은 Unix 운영체제의 파일 시스템, 프로세스 제어 메커니즘, 파이프라인 명령어들의 진화 과정을 중심으로 초기 개발 단계의 역사를 설명한다.

1968년에서 1969년까지 Bell Laboratories의 컴퓨터 과학은 Multics의 실패로 인해 너무 느려해결되지 못하던 많은 문제가 있었다.

1969년, 우리는 Multics의 대체품를 찾으려 노력했지만 너무 큰 돈이 필요했다. Thompson, R.H.Canaday, Ritchie는 후에 Unix의 심장이 되는 파일 시스템의 기본적인 디자인을 개발했지만 많은 문제들이 존재했다. 그는 또한 Multics에 관해 제안한 파일시스템 디자인에 대한 효율과 프로그램의 페이징에 관해 시뮬레이션 했지만 곧 폐기되었다.

또한 그는 우주선을 움직이며 태양계의 주요 물체들의 움직임에 관한 시뮬레이션인 'space travel'이라는 게임을 개발했지만 게임의 디스플레이 상태가 너무 변덕스러웠고 사용자가 커맨드를 입력해 제어해야 했기에 힘들었고 너무 비쌌다. 그래서 Thompson은 PDP-7 컴퓨터로 바꿨고 전체 시스템은 Graphic-II 터미널로 GECOS와 PDP-7으로 연결을 위해 어셈블리 언어로 개발했다.

게임은 매력적이었지만 크게 사용되진 않았다. 곧 Thompson은 paper file system(chalk file system)을 구체화했다. 특히 프로세스의 개념, user-level utilities인 복사, 프린트, 삭제, 파일 수정 방법들, 단순한 command interpreter(shell)를 생각해냈다. 지금까지 모든 프로그램들은 GECOS를 사용했고 파일들은 paper tape를 이용해 PDP-7으로 옮겨졌다. 잘 작동하지는 않았지만, 1970년에 Brian kernighan이 Unix라는 이름을 제안했고 우리가 아는 운영체제가 탄생했다.

PDP-7 Unix file system

구조적으로, PDP-7 Unix의 파일시스템은 현재의 시스템과 비슷하게 i-list, 디렉토리, 기기를 표현하는 특별한 파일들을 갖고 있었다.

Read, write, open, create, close 같은 중요한 파일 시스템 콜들은 처음부터 있었지만 PDP-7은 word-address 기계였기 때문에 I/O의 단위가 바이트가 아닌 단어였고 터미널의 프로세스를 erase하고 kill하는 프로세스가 부족했다. 오직 shell과 editor만 erase나 kill process를 사용했다.

PDP-7 파일 시스템은 path name이 없었고, 시스템으로의 각각의 파일 이름 인자들은 '/'이 없었다. Unix 상에서의 링크들은 존재했기 때문에 path names들이 없어도 됐다.

link는 콜을 이런 형식으로 받았다. link(dir, file, newname)

dir은 현재 디렉토리에서의 directory file, file은 그 디렉토리에 존재하는 객체, 그리고 newname은 현재의 디렉토리에 추가될 그 링크의 이름이었다. dir은 현재의 디렉토리에 존재해야 했기에, 현재의 디렉토리에 대한 링크가 가능했음이 분명하다. PDP-7 Unix 파일 시스템은 방향이 없는 그래프 형태였다.

모든 유저들이 모든 디렉토리에 대한 링크를 만들지 않아도 되게 하기 위해, dd라는 디렉토리가 있었다. 이 디렉토리는 각 유저의 디렉토리에 대한 entry를 가지고 있었다. 이 체계는 어려워서 하위 디렉터리를 실제로 사용하지 않게 했고 시스템이 실행되고 있을 때 디렉토리를 생성할 방법이 없었다. 모든 디렉토리는 재생되지 못하게 하기 위해, paper tape로부터 파일시스템을 재생성할

때 만들어졌다.

dd 방식은 몇 개의 인자를 받고 순서대로 현재의 디렉토리를 각각의 이름을 가진 디렉토리로 변환했다. 'chdir dd ken'은 ken 디렉토리로 변환할 것이다.

가장 큰 불편함은 configuration을 바꾸는 일이었는데 디렉토리들과 특별한 파일들은 둘 다 disk가 재생성됐을 때 만들어지기 떄문에 기기의 코드들이 시스템 전체에 걸쳐 퍼져 있어 새로운 기기를 설치하는 것은 힘들었다.

이 파일시스템 코드는 단순했다. Multi-program이 없어 오직 하나의 프로그램만이 메모리에 존재했고 컨트롤은 확실한 swap이 요청되었을 때만 프로세스들 간에 이동했다. 단순히 간단함 때문에 기피됐던 것은 아니었다. PDP-7에 부착된 디스크는 2ms마다 하나의 18bit 단어를 전송했다. 그에 반해 PDP-7은 1ms의 메모리 사이클을 갖고 있었고 대부분의 instruction들은 2사이클을, 간접적으로 전달된 instruction들은 3사이클을 소요했다. 기계에 index 레지스터가 없어서 간접 요청은 자주 일어났고 DMA 컨트롤러는 instruction 처리 동안 메모리에 접근할 수 없었다. 전송 중에 간접 요청된 instruction들이 실행되면 디스크가 에러들을 발생시켰다. 그래서 디스크가 돌 때 일반적 시스템 코드도 실행될 수 없었고 컨트롤도 유저에게 돌아갈 수 없었다. 항상 실행 되어야할 clock과 터미널들의 interrupt 루틴은 indirection을 피하기 위해 이상한 방식으로 코딩 돼야 했다.

Process control

Process control이란 프로세스들이 생성되고 사용되어지는 메커니즘이다. 현재는 fork, exec, wait, exit을 콜해 이런 메커니즘이 작동한다.

요즘, shell에서 command들이 실행되는 방식은 아래와 같다.

1) Shell이 터미널로부터 command line을 읽어온다.
2) fork를 이용해 child process를 생성한다.
3) Child process는 파일로부터 command를 call하기 위해 exec를 사용한다.
4) 그 동안, parent shell은 child process가 exit을 call해 terminate 전까지 wait으로 기다린다.
5) Parent shell은 다시 step1로 돌아간다.

PDP-7 Unix에 process들은 존재했다. Fork, wait, exec은 없었고 Exit은 있었지만, 의미가 조금 달랐다. Shell의 main loop는 아래와 같았다.

1) Shell은 열린 모든 파일들을 닫고, standard 입출력을 위해 terminal special file을 연다.
2) 터미널로부터 command line을 읽어온다.
3) 커맨드가 가리키는 파일을 link해 파일을 열고 link를 삭제했다. 메모리의 상단에 작은 bootstrap 프로그램을 복사해 jump했다. 이 bootstrap 프로그램은 shell code를 통해 파일을 읽었고 다시 커맨드의 첫번째 위치로 jump했다.
4) 커맨드는 작동한 후, exit을 콜해 terminate 되었다. exit 콜은 시스템이 terminate된 커맨드를 shell의 복사본을 통해 읽게 했다. 다시 step1로 돌아갔다.

이 때는 background process들이나, shell command file들이 지원되지 않았지만 IO redirection(<,>)은 곧 개발되었다.

각 터미널당 하나의 프로세스라는 개념의 process control은 많은 interactive system들과 비슷하다. 일반적으로 이런 시스템들은 computation 분리와 커맨드 파일들과 같은 유용한 기능들을 실행하기 위해 특별한 메커니즘을 필요로 했다. 또한 path name이 없었기 때문에 /dev 디렉토리가

존재하지 않았고 shell은 각각의 command 이후에 재실행되어야 했기 때문에 command를 통해 메모리를 재확보할 수 없었다. 그래서, 그 이상의 파일시스템이 필요해졌다. 각각의 디렉토리는 그 파일을 연 프로세스의 터미널의 special file을 위한 entry인 tty를 담고 있어야 했다. 만약 entry가 없는 디렉토리를 변경했다면, shell은 의미없이 반복문을 돌게 될 것이다.

현대적인 형식의 process control은 곧 디자인되었지만 문제가 있었다. 예로는 fork과 exec 함수들의 분리가 있다. 새로운 process들을 생성할 때 실행할 process를 위한 program을 명시해야 한다. Unix에서 fork된 process들은 exec를 실행하기 전까지는 parent로서 같은 프로그램을 계속 실행한다. 시스템은 이미 multiple process를 처리할 수 있었고 process table이 존재했고, main memory와 디스크 사이에서 swap 되었다. fork의 초기 실행이 필요했던 것들은 아래와 같다.

1) Process table의 확장
2) 이미 존재하는 초기 swap IO를 사용하여 disk의 swap area로 현재 process를 복사하고 process table에 몇몇의 수정을 하는 fork call의 추가

PDP-7의 fork의 콜은 운영체제와 user program들에서 많이 변화가 요구됐고 긴 어셈블리 코드가 필요했다. 그러나 결합된 fork-exec는 더 복잡했다.

새로운 버전에서 exit call은 process table entry를 정리하고 control을 포기하기만 하면 됐다.

wait이 된 초기의 기능은 현재 체계에 비해 더 단순했다. process들 간에 one-word message를 보냈다.

smes의 process는 receiver 와의 어떤 선례의 관계도 가질 필요가 없었다. Message들은 queue되지 않고 sender는 receiver가 message를 읽기 전까지 delay 되었다.

Message 기능은 커맨드를 실행할 process를 생성한 후에 parent shell은 새로운 process에 smes를 이용해 message를 보냈다. Command가 terminate되었을 때, shell의 block된 smes 콜은 target process가 존재하지 않는다는 error를 반환한다. shell의 smes는 사실 wait과 동일해졌다.

Message는 보편적으로 protocol들은 각 터미널의 프로그램 초기실행과 shell들 간에 사용되었다.

메커니즘에서의 버그도 있었고 현대의 wait콜은 이보다 목적에 직관적이어서 대체되었다고 볼 수 있다. command process가 다른 process들과 소통을 위해 message를 사용하려 할 때, 이는 shell의 동시화를 방해했다. Shell은 receive하지 않은 message보내는 것에 의존했다. 만약 command가 rmes를 실행했다면, shell의 허위의 message를 받게될 것이고, 이는 command line이 terminate된 것처럼 다른 입력 줄을 읽게 할 것이다.

어쨌든, 새로운 process control 체계는 몇몇의 굉장히 가치 있는 특징들을 만들었다. 예로는, 분리된 process들과 command로서의 shell의 반복적 사용이 있다. 대부분의 시스템들은 'batch job submission' 같은 기능과 상호 사용되다 분리된 파일들을 위한 특별한 command interpreter를 지원해야 한다.

Multiple-process 도입 후 얼마 지나지 않아 chdir 작동에 문제가 생겼다. Chdir은 터미널의 process의 현 디렉토리를 수정하기 위해 프로세스의 디렉토리를 바꾸지만 이 프로세스는 바로 terminate 되었고, parent shell에 아무 효과도 없었다.

시스템과 새로운 프로세스 컨트롤 체계 간의 문제는 오랜 시간이 지나 나타났다. 각각의 파일 열기에 관련된 read/write 포인터는 파일을 연 process 안에 저장되어 있었다. 이 정렬의 문제는 우리가 command file들을 사용했을 때 명백해졌다. 아래와 같은 내용을 포함하는 간단한 command file인 ls who는 sh comfile > output로 작동된다. 그 후에는,

1) main shell은 standard 출력을 receive하기 위해서 output을 열고, shell을 반복적으로 실행시키는 새로운 process를 만든다.
2) 새로운 shell은 ls를 실행하기 위해서 다른 process를 생성한다. 이것은 output 파일에 올바르게 쓰고, terminate한다.
3) 다음 command를 실행하기 위해 다른 process가 생성된다. 하지만, output을 위한 IO 포인터는 shell에서 복사되고, 아직 0의 값을 갖는다. 왜냐하면 shell은 output에 아무것도 쓰지 않은 상태고 IO 포인터들은 process들과 관련있기 때문이다. 그래서 who의 결과가 overwrite하고 진행중인 ls 커맨드의 결과를 파괴한다.

해결하려면 열려있는 프로세스들의 파일들의 io 포인터들을 독립적으로 저장하기 위해 새로운 system table을 만들어야 했다.

IO Redirection

'>', '<'을 사용하는 IO redirection은 PDP-7 Unix 시스템의 꽤 이른 시기에 나타났다. Multics는 여러가지 기기들, 파일들, 심지어 특별한 stream-processing 모듈로 동적으로 redirect될 수 있는 IO stream들을 구현하는 더 보편적인 IO redirection 매커니즘을 가지고 있었다.

PDP-11의 출현

1970년대 초기에, PDP-7은 좋은 프로그래밍 환경을 제공했지만 구식이었다. 우리는 현재에 'word-processing system'이라고 불리우는 텍스트를 수정하고 formatting하는 시스템을 만들기 위해 PDP-11의 구입을 제안했고 5월에 주문되었다.

Processor은 여름이 끝날 때쯤 도착했고 그 동안, 가장 기본적인 core-only 버전의 유닉스는 PDP-7에서 cross-assembler를 사용해 쓰여졌다.

첫 PDP-11 system

PDP-11의 unix의 초기 버전은 multi-programming이 없었다. 코어에서는 오직 한 user program 만 존재할 수 있었다. 반면에, 유저의 인터페이스에 아주 중요한 변화가 있었다. Path names로 채워진 현재 디렉토리 구조가 있었다. 현대 형식의 exec와 wait, 터미널을 위한 편리 기능인 character-erase, line-kill같은 process들도 있었다. 24K 바이트의 코어 메모리가 있었고, 1K 블록들의 디스크가 있었다. 파일들은 64K 바이트들로 제한되었다.

PDP-11을 주문했을 때, word processing만을 하는 시스템을 갖는 것이 자연스러워 보였지만 PDP-7 Unix의 유용함은 PDP-11 Unix를 개발하는 것이 적합하다고 생각하게 만들었다. 그래서 roff text formatter를 PDP-11 어셈블러 언어로 옮겨썼다. 편집자와 formatter를 확보했고, 특허 적용을 위해 특허 부서에 text-processing 제안했다. 우리가 제안한 주요한 이점은 Teletype 모델의 37 터미널들을 지원하고 필요로 했던 대부분의 수학 기호들을 출력해낼 수 있다는 것이었다. 또한 roff에 줄번호가 있는 페이지들을 생성할 수 있었다.

Unix는 보통의 하드웨어에서 흥미로운 서비스들을 지원할 수 있다는 좋은 평판을 갖고 있었고 이 기간은 이익/장비 비율에서 아주 높은 수치를 기록했다. 하나의 메모리 보호도 없었고 하나의 0.5MB 디스크에서 새로운 프로그램의 모든 테스트들은 불안정했다. 왜냐하면 시스템을 쉽게 충돌할 수 있었고, 이는 몇 시간 마다 타이핑 직원들은 DECtape에 더 많은 정보를 밀어넣는다는 것을

의미했다. 디스크가 매우 작았기 때문이다.

특허 부서가 Unix를 채택하고 Laboratories에서 승인 받았고 우리는 처음 만들어진 PDP 11/45 시스템들 중 하나를 만들어 신뢰를 얻었다.

Pipes

운영체제와 command 언어들의 문화에 Unix의 가장 널리 인정받는 기여 중 하나는 command의 pipeline에서 사용되는 pipe이다. Pipeline은 단순히 coroutine의 특정한 하나의 형태였다. Dartmouth Time-sharing System의 'communication files'는 Unix pipe들과 거의 근접하게 유사한 작업을 하지만, 완벽히 구현됐다고는 볼 수 없었다.

Pipe는 1972년에 PDP-11 버전이 사용된 후, Unix에 등장했다. Pipe가 개발되기 몇 년 전에, 그는 command들이 왼쪽과 오른쪽의 연산자들이 input과 output을 나타내도록 이진 연산자로서 생각되어야 한다고 제안했다. 그러므로 'copy' 기능은 아래와 같이 쓰였다.

inputfile copy outputfile

Pipeline을 만들기 위해, command operator들은 쌓아졌어야 했다. input을 구별해내고, 깔끔하게 순서 메기고, 결과를 다른 라인에 프린트하기 위해 이렇게 쓰여야 했다.

input sort paginate offprint

현대의 시스템에서는 이와 같다. 'sort input | pr | opr'

Infix notation이 너무 급진적이라는 문제가 있었다. 명령 인자들을 input이나 output 파일에서 구별해낼 수 없었고 하나의 input, output 모델의 command 실행은 너무 제한적이었다.

얼마 후, pipe들은 새로운 표기법으로 운영체제에 설치되었다. IO redirection과 같은 철자들을 사용해 이와 같이 쓰였다. 'sort input >pr>opr>'

'>' 뒤에 나오는 것은 output의 redirection을 다시 그 파일로 특정지어주기 위한 파일명이거나, 이전의 command의 output이 input으로 간주되는 명령이었다.

곧 이 표기의 문제들이 나타났다. '>' 이후의 문자열은 공백들로 범위가 정해졌기에 아래의 예시처럼 pr에 인자를 주기 위해서는 따옴표를 써야 했다.

sort input >"pr -1">opr>

두 번째 문제점은, 일반성을 주기 위해, pipe 표기법은 '<'을 '>'와 같이 input redirection으로 받아들였다. 즉, 이렇게도 쓰일 수 있다는 말이었다.

opr <pr<"sort input"< 이거나 pr <"sort input"< >opr>

'<'이나, '>'를 사용하는 pipe 표기법은 얼마 후 Pipeline의 구성요소를 분리하는 특유한 연사를 사용하는 현대의 표기법으로 바뀌었다. 물론, 이에도 제한점들이 있었다. 여러 개의 redirect된 input과 output들이 콜된 상황들에서도 직선 형태를 유지했다.

Multics는 기기에서 또는 기기로, 파일이 source나 sink로 작용하는 방법으로 module processing을 통해서 IO stream들이 direct되는 메커니즘을 제공했다. 그러므로 Multics에서 stream-splicing은 Unix pipe의 직접적인 선구자였다고 볼 수 있다.

High-level languages

원래의 PDP-7 Unix 시스템을 위한 모든 프로그램들은 매크로가 존재하지 않는 어셈블리 언어로 쓰여졌다. loader나 link-editor도 없었다, 그래서 모든 프로그램들은 그 자체로 완벽해야만 했

다. 처음 개발된 언어는 McIlroy의 McClure's TMG 버전이었다. Thompson은 BCPL 언어에 영향을 받은 B언어와 컴파일러를 만들었다. 컴파일러는 단순한 단순한 interpretive 코드를 만들어냈다. 이 언어는 조금 느렸지만, 편리했다. Regular system call들에 대한 인터페이스가 사용 가능하게 되었다. PDP-11을 위한 PDP-7 B cross compiler는 B로 쓰여졌고, PDP-7 자체를 위한 B 컴파일러는 TMG를 B로 옮겨쓴 것이다.

PDP-11이 도착하고 B를 사용하기 시작했다. 사실, multi-precision 'desk calculator' 프로그램인 dc의 버전은 잘 돌아가던 PDP-11에서 처음으로 돌려진 프로그램 중 하나였다. 어셈블러가 아닌 B로 운영체제를 재작성하자는 이야기가 나왔지만 interpretive 코드는 느렸고 word-oriented인 B 언어와 byte-address 인 PDP-11가 불일치했다.

1971년도에 C언어 개발이 시작되었다. 운영체제 커널이 C언어로 재작성됐을 때, 가장 중요한 변화는 1973년도의 multi-programming의 시작이었다. 시스템이 현대의 형태를 띄게 된 시점이다. 시스템의 내부적 구조는 훨씬 보편적으로 수정되었고 이를 통해 C언어가 시스템 프로그래밍을 위한 보편적인 도구로서 아주 유용하다는 점이 증명됐다.

오늘 날, 어셈블러로 쓰여진 Unix의 중요한 프로그램은 어셈블러 자체밖에 없다. Unix의 성공은 가독성, 수정 가능성, 소프트웨어의 이식성, 즉 high-level 언어의 표현들로부터 나왔다.

Takeaway

이 논문은 Unix 운영체제의 전체적인 개발 역사를 설명하고 있다. 주요 내용들로는 파일 시스템, 프로세스 제어 메커니즘, 파이프라인에 대한 개발 내용이 있다.

초기에도 파일 시스템에서 read, write, open, create, close와 같은 현재와 비슷한 콜들이 존재했다. 하지만 PDP-7 기계는 word-address 기계였기 때문에 byte 단위로의 처리가 불가능했고 erase와 kill 프로세스가 사용이 불가능했다. 또한, link가 존재했기 때문에 path name들이 존재하지 않았다. 즉, PDP-7 Unix의 파일 시스템은 방향이 없는 그래프 형태였다고 볼 수 있다.

PDP-7 Unix에도 fork, wait, exec, exit의 프로세스들이 존재했다. 하지만 이는 computation 분리 밑 커맨드 파일들과 같은 기능을 사용하기 위해선 추가적인 과정이 필요했고 부족한 파일시스템 때문에 문제가 많았다. fork와 exec 함수를 분리하려 했지만 fork 콜은 운영체제와 user program들 사이에서 많은 변화가 필요했다. 그러나 결합된 fork-exec는 더 복잡했다. 새로운 버전의 exit은 process table entry를 정리하고 control을 포기하기만 하면 됐고, 현재의 wait이 된 당시의 기능인 smes 현재보다 단순했다.

저자의 팀은 Teletype 모델의 37 터미널들을 지원하고 필요로 했던 대부분의 수학 기호들을 출력해낼 수 있는 roff text formatter를 PDP-11 어셈블러 언어로 옮겨 써 첫 PDP-11 Unix를 개발했고 승인 받았다.

초기 Pipeline은 단순히 coroutine의 특정한 하나의 형태였다. 하지만 infix notation이 너무 급진적이라는 문제를 해결해 많은 수정을 거쳤다.

원래의 PDP-7 Unix는 어셈블리 언어로 만들었지만, Thompson은 B언어와 컴파일러를 만들었지만 느렸고 word-oriented인 B언어와 byte-address인 PDP-11이 불일치했다. 그래서 C언어의 개발이 1971년도에 개발되었다. 이와 함께 multi-programming이 시작되며 시스템이 현대의 형태를 띄게 되었다. Unix의 성공은 high-level 언어로부터 나왔다고 볼 수 있다.

논문이지만 Unix라는 운영체제의 개발 과정에 대해 읽기 쉽게 설명되어 있어 매우 유용한 시간이었다. 많은 프로세스 및 기능들의 변천사에 대해 초점이 맞춰져 있었지만, 아직 운영체제에 대한 깊은 공부를 하지 못해본 나에게는 그 이외에 이해에 필요했던 배경 지식들에 대한 공부까지 하게 하며 매우 흥미로운 시간이었다. High-level 언어인 파이썬을 주로 사용하던 나에게 이런 low-level에 대한 지식이 많이 부족해 항상 걱정이었지만, 이 논문을 읽으며 운영체제에 대한 근본적인 지식에 대한 공부를 시작하기에 앞서 매우 좋은 선행 공부를 할 수 있었다.

가장 놀라웠던 사실은 multi-programming의 중요함이었다. 초기 개발 단계에서의 많은 문제점들과 현재에 비한 낮은 성능들은 multi-programming의 부재에서 나왔다. 1971년도에 C언어의 개발과 함께 시작된 multi-programming의 개발은 읽으면서도 개발에 대한 경외심을 감출 수 없었다. 현재 내가 컴퓨터 공학과 공부를 하며 접하게 되는 많은 지식들과 학문들의 근본을 들여다본 시간이었다.

공부를 하게 되며 배우게 될 프로그래밍 기법들이나, 함수들에 대한 기본 지식을 알게 되어 앞으로 그저 표면적인 코딩이 아닌 그 이상의 깊은 공부를 할 수 있게 된 듯한 기분이 들었다. 어떤 기능이 개발되었지만, 어떤 문제점이 있었고, 그래서 어떤 방향의 해결이 필요했다라는 자세한 서술이 논문의 모든 부분에 포함되어 있어 이해도 쉬웠으며 그만큼 많은 공부가 된 듯한 글이다.