유닉스 프로그래밍 및 실습

1장. 소개와 핵심개념

개요

- 시스템 소프트웨어
  - 저수준에서 동작
  - 커널/핵심 라이브러리와 직접 인터페이스
  - 예:
    - 쉘, 텍스트 편집기, 컴파일러, 디버거, 유틸리티, 시스템 데몬
- 리눅스
  - 현대적인 유닉스 계열 시스템
  - 리누스 토발즈 + 느슨한 해커 공동체
  - 유닉스가 추구하는 목표와 이상은 공유하지만, 동일하지는 않다
    - 추가적인 시스템 호출
    - 다른 동작 특성
    - 새로운 기능
  - 실용적인 쪽에 집중

1. 시스템 프로그래밍 (1)

- 전통적으로 모든 유닉스 프로그래밍은 시스템 프로그래밍이다.
  - 역사적으로 고수준 추상화가 없음
    - X 윈도우 프로그래밍의 경우 프로그래머가 핵심 유닉스 API에 바로 노출됨
- 응용 프로그래밍과의 차이
  - 시스템 프로그래머는 프로그램이 동작하는 기반 하드웨어와 운영체제를 제대로 알고 있어야 함
  - 프로그램을 작성하는 '스택' 수준 차이
- 응용프로그래밍이 시스템 프로그래밍과는 멀어져 고수준 프로그래밍으로 이동하는 시류
  - 웹 소프트웨어(Java Script, PHP 등)
  - 관리되는(managed) 코드 (C#, Java)
- 그러나 시스템 프로그래밍의 종말은 오지 않는다
  - 누군가는 자바스크립트 인터프리터나 C# 런타임을 작성해야
- 유닉스와 리눅스 코드의 대부분은 여전히 시스템 수준에서 만들어짐
- 본 교재의 범위
  - 커널 개발이나 드라이버 개발, 네트워크 프로그래밍은 배제 (그 자체가 한 권의 교재로 다뤄져야 함)
  - 사용자 영역에 위치한 시스템 프로그래밍에 초점
    - 커널 위에 있는 모든 프로그래밍 관련 내용

1. 시스템 프로그래밍 (2)

- 시스템 호출(system call)
  - 사용자 영역에서 시작해서 시스템 핵심인 커널로 들어가는 함수 호출
    - read(), write()
    - get_thread_area(), set_tid_address()
  - 다른 운영체제에 비해서 훨씬 적게 구현
    - i386 : 대략 300개
    - MS Windows : 수천개
  - 아키텍처(alpha, i386, power PC)에 따라 다를 수 있지만 90%가 넘는 시스템 하위 공통집합을 가지고 있음
  - 본 교재는 90%가 넘는 공통집합을 다룸
- 시스템 호출 실행
  - 보안과 안정성 때문에 사용자 영역 응용이 직접 커널 코드를 실행하거나 커널 자료를 접근할 수 없음
  - 대신 시스템 호출을 원한다는 '시그널'을 커널에 전달
  - 매개변수 전달
    - i386의 경우 5개의 레지스터에 순서대로 담고, 마지막 6번째 레지스터에는 사용자 영역에 있는 버퍼 주소를 담아 전달
  - 실행 방식은 아키텍처마다 다를 수 있음

1. 시스템 프로그래밍 (3)

- C 라이브러리
  - 유닉스 응용을 구성하는 핵심
  - 현대적인 리눅스 시스템에는 GNU libc(glibc)가 제공됨 (지-리브-씨 또는 글리브씨)
    - 표준 C 라이브러리 구현 + 시스템 호출 래퍼, 쓰레드 지원, 기본 응용 기능 제공
- C 컴파일러
  - gcc(GNU Compiler Collection)
    - 과거에는 GNU C Complier를 의미했으나 이제는 더 많은 언어 지원

2. API와 ABI (1)

- 현재와 미래에 지원을 약속한 모든 시스템에서 프로그램이 돌도록 만드는 작업에 관심
  - API(Application Programming Interface)
  - ABI(Application Binary Interface)
- API
  - 원시 코드 수준에서 다른 부분과 인터페이스 하는 방식을 정의
  - 함수 형태인 표준 인터페이스 집합을 공개하는 방법으로 추상화 제공
  - '계약'이라고 표현하지만 일반적으로 API 사용자는 PAI 구현에 영향을 미치지는 못함

2. API와 ABI (2)

- ABI
  - 특정 아키텍처에서 동작하는 소프트웨어 사이에서 필요한 저수준 이진 인터페이스
  - 응용과 응용, 응용과 커널, 응용과 라이브러리 사이에서 일어나는 상호 작용 정의
  - 재컴파일 과정 없이 동일 ABI를 사용하는 시스템 사이에서 목적 코드가 동작하도록 이진 호환성 보장
  - 호출 관례, 레지스터 활용, 시스템 호출, 링크, 라이브러리 동작방식, 이진 목적파일 형식 등 정의
  - 다양한 운영체제에서 특정 아키텍처용 단일 ABI를 정의하려는 시도가 몇 차례 있었으나 결실을 맺지 못함
  - 리눅스는 상황에 맞는 독자 ABI 정의하려는 경향(아키텍처마다 ABI가 제각각)

3. 표준 (1)

- 유닉스 프로그래밍 기초는 수십년간 그대로 유지
- 유닉스는 동적
  - 동작 방식 변화, 기능 추가
  - 시스템 인터페이스를 공식 표준으로 성문화
- 리눅스는 어떤 표준도 준수하지 않지만, POSIX(Portable 와 SUS(Single UNIX Specification)을 따르려고 노력

3. 표준 (2)

- POSIX 역사
  - 1980년대 중반 IEEE에서 유닉스 시스템용 시스템 인터페이스 표준화 시도
  - 리처드 스톨만 POSIX 이름 제안
  - 1988년 IEEE Std. 1003.1-1988(POSIX 1988)
  - 1990년 IEEE Std. 1003.1-1990(POSIX 1990)
  - 실시간 옵션 IEEE Std. 1003.1b-1993(POSIX 1993 또는 POSIX 1b)
  - 스레드 지원 옵션 IEEE Std. 1003.1c-1995(POSIX 1995 또는 POSIX 1c)
  - 2001년 옵션까지 포함한 단일 표준 IEEE Std. 1003.1-2001(POSIX 2001)
  - 2004년 최신 표준 IEEE Std. 1003.1-2004(POSIX.1)

3. 표준 (3)

- SUS 역사
  - 1980년대 후반 1990년대 초반의 유닉스 전쟁
    - Open Group = OSF(Open Software Foundation) + X/Open
    - 오픈 그룹에서 SUS 발표
      - 인기 이유 : POSIX에 비해 비용이 저렴
      - 최신 POSIX 포함
  - 1994년 첫 번째 버전(SUSv1) 발표 – UNIX95 인증 로고
  - 1997년 두 번째 버전 발표 – Unix98 인증 로고
  - 2002년 세 번째 버전 발표 – UNIX03 인증 로고
    - IEEE Std 1003.1-2001과 다른 표준을 개선하고 결합

3. 표준 (4)

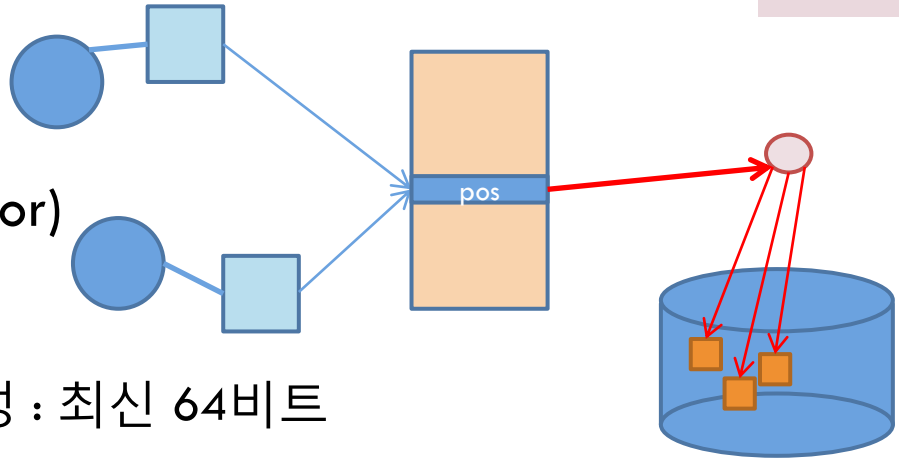
- C 언어 표준
  - 1978년 데니스 리치, 브라이언 커닝헌 “The C Programming Language” 출간 (K&R C)
  - 1983년 ANSI(American National Standards Institute)에서 표준화를 위한 위원회 구성
  - 1989년 ANSI C 탄생
  - 1990년 ISO C90 승인
  - 1995년 개선된 C 언어 ISO C95 발표
  - 1999년 ISO C99 발표
    - 인라인 함수, 새로운 자료 타입, 가변 길이 배열, C++ 주석, 새로운 라이브러리 함수 포함
  - gcc에서 C99를 이용하여 컴파일하고자 하는 경우
    - gcc abc.c -std=c99

3. 표준 (5)

- 리눅스와 표준
  - POSIX와 SUS 준수 노력
    - 문서화된 인터페이스 제공
    - 요구사항에 맞춘 동작방식
  - 공식적인 인증 작업을 안 거쳤으므로 호환이라고 주장하지는 못함
  - gcc C 컴파일러는 C99 지원
  - LSB(Linux Standard Base)
    - 리눅스 파운데이션 후원 아래 여러 업체가 공동으로 벌이는 프로젝트
    - POSIX, SUS를 확장하여 독자적인 여러 표준 추가
- 이 책과 표준
  - 리눅스 커널 (2.6), gcc C 컴파일러(4.2), C 라이브러리(2.5)를 사용하는 시스템 프로그래밍 방법을 구체적으로 다룸

4. 리눅스 프로그래밍 개념 (1)

- 파일과 파일시스템
  - 파일
  - 열기/읽기/쓰기/닫기
  - 파일 기술자(fd: file descriptor)
  - 일반 파일
    - '위치' (location / offset)
    - 위치값 : C 타입 크기로 결정 : 최신 64비트
    - 파일 크기 = 길이
    - 파일은 두 번 이상 열 수 있다
      - 공유 가능
      - 동시 접근 결과는 개별 연산 순서에 의해 달라짐(예측 불가능)
      - 충분히 동기화되도록 스스로 조율해야 함
    - inode(information node)
      - 파일에 관련된 메타정보 저장
    - ino(i-number)



4. 리눅스 프로그래밍 개념 (2)

- 파일과 파일시스템(계속)
  - 디렉토리와 링크
    - 디렉토리
      - 이름을 inode 번호에 사상
      - 링크 : 이름과 inode 번호 쌍
    - 계층구조
    - dentry 캐시 활용
      - 디렉토리 결정 결과 저장, 시간적인 지역성을 활용하여 탐색속도 개선
    - 절대경로 / 상대경로
    - 특수한 시스템 호출 집합 활용
    - 링크 추가 / 삭제

.	2056
..	1567
abc.c	2045
def.l	6731
ghi.c	2045
mydoc1	3112

4. 리눅스 프로그래밍 개념 (3)

- 파일과 파일시스템(계속)
  - 하드링크
    - 다중 링크는 동일한 inode를 여러 이름으로 사상
    - 파일 삭제에 대한 고려
      - 링크 카운터(reference count)가 0에 도달해야 실제 삭제가 일어남
  - 심볼릭 링크(symbolic link, symlink)
    - 파일시스템을 가로지르는 좀더 단순하지만, 덜 투명한 링크 허용
    - 하나의 일반 파일
      - 링크로 연결할 파일의 완전한 이름을 포함하는 독자적인 inode와 자료
      - 디렉토리, 존재하지 않는 파일도 가리킬 수 있음
    - 하드링크보다 많은 부하
      - 심볼릭 링크와 링크로 연결한 파일 둘 다 다루어야 함
    - 투명성이 떨어짐
      - 단축 기능으로 이용 가능

4. 리눅스 프로그래밍 개념 (4)

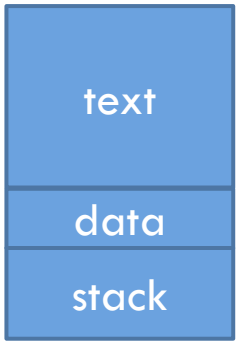
- 파일과 파일시스템(계속)
  - 특수 파일
    - 블록 디바이스, 문자 디바이스, named pipe, 유닉스 도메인 소켓
    - 모든 것이 파일이라는 사고 들로 추상화한 것
    - 문자 디바이스
      - 바이트로 구성된 선형 큐
      - 키보드
    - 블록 디바이스
      - 바이트 배열로 접근
      - seek
      - 임의 접근
    - Named pipe(FIFO)
      - IPC(InterProcess Communication) 메커니즘의 일종
      - 일반 파이프는 메모리 상에 존재 (부모-자식 프로세스간 통신만 가능)
      - 이름 붙은 파이프는 FIFO 특수 파일을 거쳐 일반 파이프처럼 접근
      - 서로 무관한 프로세스 사이에 통신 가능
    - 유닉스 도메인 소켓
      - 소켓 : IPC(InterProcess Communication) 메커니즘의 일종
      - 파일시스템 상의 특수 파일(소켓)을 이용하여 통신

4. 리눅스 프로그래밍 개념 (5)

- 파일과 파일시스템(계속)
  - 파일시스템과 이름 공간
    - 단일 전역 이름공간(namespace) 이용
    - 파일시스템
      - 정형적이고 유효한 계층구조로 이루어진 파일과 디렉토리를 모아놓은 집합
      - 파일시스템을 개별적으로 추가하거나 해제 가능 (마운트/언마운트)
      - 물리적 파일시스템(디스크, CD, 플로피), 가상파일시스템(메모리), 네트워크 파일시스템 지원
    - 섹터, 블록
    - 프로세스 단위 이름공간 지원
      - 개별 프로세스가 시스템에 존재하는 파일과 디렉토리 계층구조를 독자적으로 볼 수 있음

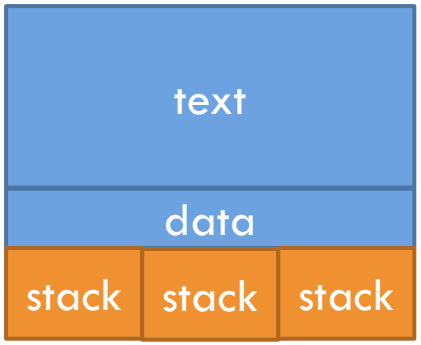
4. 리눅스 프로그래밍 개념 (6)

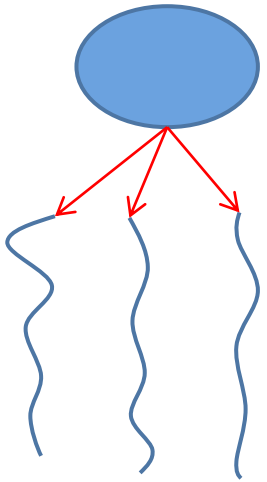
- 프로세스
  - 실행 중인 목적코드
    - 활성화되어 있으며, 살아있고, 프로그램을 실행
    - 자료, 자원, 가상화된 컴퓨터 포함
    - 커널에서 이해하는 실행파일 형식(ELF)으로 만들어진 목적코드로 삶을 시작
      - 메타 자료, 다중 섹션 포함
        - 텍스트 : 코드
        - 자료 : 초기화된 전역 변수
        - bss(block started by synblo 또는 block storage segment)
          - 초기화되지 않은 전역 자료 포함
          - 목적코드에서 초기화되지 않은 변수 목록 유지하고, 커널에 올라오는 경우 0인 페이지에 사상
      - ELF의 경우 (재배치 불가능한 심볼을 포함하는) 절대 섹션과 미정의 섹션 포함
    - 프로세스와 자원
      - 타이머, 대기 중인 시그널, 열린 파일, 네트워크 연결, 하드웨어, IPC 매커니즘 등
    - 선점형(preemptive) 멀티태스킹 / 가상 메모리(Virtual memory) 지원
      - 가상화된 프로세스와 가상화된 메모리 제공
        - 각각이 전체를 혼자 독점하는 것처럼 동작
        - 프로세스마다 다른 주소공간에서 동작



4. 리눅스 프로그래밍 개념 (7)

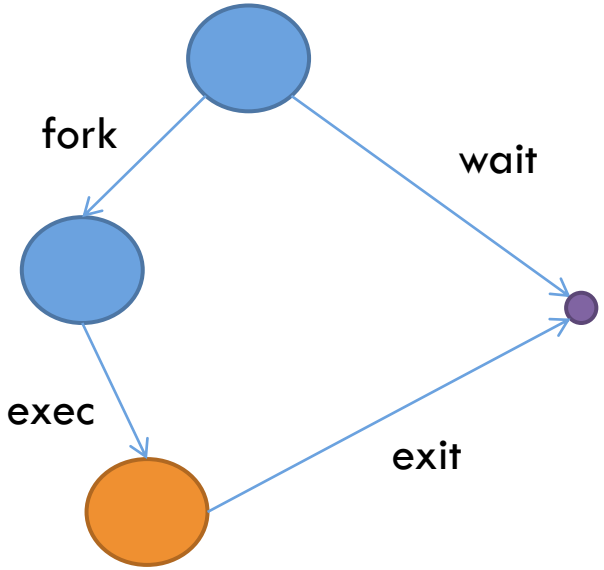
- 프로세스(계속)
  - 스레드
    - 프로세스는 실행 스레드를 하나 이상 포함
    - 프로세스 내부에서 실행하는 활동 단위
    - 코드를 실행하고 프로세스 동작 상태를 유지하는 책임을 맡은 추상화
    - 전통적인 유닉스
      - 단일 스레드지만, 역사적인 단순성, 빠른 프로세스 생성시간, 견고한 IPC 메커니즘 지원
    - 스레드는 스택, 프로세서 상태, 목적코드에서의 현재 위치를 포함하고, 다른 자원은 프로세스 내의 모든 스레드가 공유
    - 리눅스 입장에서는 몇몇 자원을 공유하는 일반적인 프로세스일 뿐 – 사용자 영역에서 POSIX 1003.1c(pthread)에 따라 구현





4. 리눅스 프로그래밍 개념 (8)

- 프로세스(계속)
  - 프로세스 계층
    - pid(process id)
      - 첫 번째 프로세스 1(init)
      - 엄격한 계층구조 형성
      - 부모 – 자식
      - fork – exec – wait – exit
      - 고아 – 좀비



4. 리눅스 프로그래밍 개념 (9)

- 사용자와 그룹
  - uid(사용자 ID)
    - /etc/passwd에 저장
  - Login(1) 과정
    - 사용자가 입력한 이름과 암호가 올바르면 사용자 로그인 쉘을 실행(exec)시키고, 쉘의 uid를 사용자 uid로 변경
  - 자식 프로세스는 부모 프로세스의 uid 상속
  - uid 0(root) : 특별한 권한이 있음
  - 실제 uid : 프로세스를 시작한 사용자
    - 유효 uid: 다른 사용자 권한으로 프로세스 실행
    - 저장된 uid: 원래 유효한 uid 저장
    - 파일시스템 uid: 일반적인 유효 uid와 같으며, 파일시스템 접근 검증
  - 사용자는 하나 이상의 그룹에 속함
    - /etc/group
  - 실제 gid, 유효 gid, 저장된 gid, 파일시스템 gid
  - 전통적으로 uid와 gid 기반 접근 통제
    - 최신 보안시스템은 좀더 세밀한 접근통제 가능

4. 리눅스 프로그래밍 개념 (10)

- 접근 권한
  - 전통적인 유닉스와 동일
  - 소유자, 그룹, 그외 접근 권한 비트 집합
  - 읽기/쓰기/실행 권한
    - 일반파일
    - 특수파일: 읽기/쓰기 (실행은 무시)
    - 디렉토리: 읽기(디렉토리 목록 열거)/쓰기(새로운 링크 추가/기존 링크 삭제)/실행(디렉토리 내부로 들어가기)
  - 리눅스는 전통적인 유닉스 접근 권한외에도 ACL(Access Control List)도 지원

4. 리눅스 프로그래밍 개념 (11)

- 시그널(signal)
  - 비동기식 일방 통지 메커니즘
  - 대략 30개 정도의 시그널 구현
  - SIGKILL(예외없이 프로세스 종료), SIGSTOP(예외없이 프로세스 중단)을 제외한 다른 시그널은 제어 가능
    - 기본 동작 (종료/종료 및 코어 덤프/중단/무시)
    - 무시
    - 독자적으로 처리
  - 처리 후 인트럽트 된 위치로 귀환

4. 리눅스 프로그래밍 개념 (12)

- 프로세스간 통신
  - 프로세스 사이에서 일어나는 정보 교환과 사건 통지
  - 파이프
  - 이름 붙은 파이프
  - 세마포어
  - 메시지 큐
  - 공유메모리
  - 퓨텍스(futex) – fast userspace mutex
    - 리눅스 고유 메커니즘

4. 리눅스 프로그래밍 개념 (13)

- 헤더
  - 표준 C 계열(예: <string.h>)
  - 일반적인 유닉스 계열(예: <unistd.h>
- 오류 처리
  - errno : 오류의 구체적인 이유를 알 수 있는 변수
    - <errno.h>에 전역변수로 정의되어 있음
    - 함수 호출 직후에만 유효하다는 사실에 주의
    - 표 1-2 참조
  - errno 관련 함수
    - perror (<strio.h>)
    - strerror(), strerror_r()
      - strerror_r()이 스레드 안전 함수

5. 시스템 프로그래밍을 시작하며

실습과제

- 링크, 심볼릭 링크 관련 UNIX 명령어 정리
- 1장 errno 이용 메시지 출력 부분 완성 및 테스트
- <errno.h> 등 본문에 나온 헤더 파일 요약 정리 – 기말 과제
- perror(), strerror(), strerror_r() 요약 정리 – 기말 과제