

LG전자 BS팀

『 1일차 』: 오후

- ◆ 훈련과정명 : OS 기본
- ◆ 훈련기간 : 2023.05.30 ~ 2023.06.02



목차



『2과목』

4-5교시 :

운영 체제 개관





학습목표

- 이 워크샵에서는 운영 체제에서 제공하는 서비스 식별할 수 있다.
- 운영 체제 서비스를 제공하기 위해 시스템 호출이 사용되는 방법 설명할 수 있다.
- 운영 체제 설계를 위한 모놀리식, 계층화, 마이크로커널, 모듈식 및 하이브리드 전략 비교 및 대조할 수 있다.
- 운영 체제 부팅 프로세스 설명할 수 있다.
- 운영 체제 성능 모니터링을 위한 도구 적용할 수 있다.
- Linux 커널과 상호 작용하기 위한 커널 모듈 설계 및 구현할 수 있다.

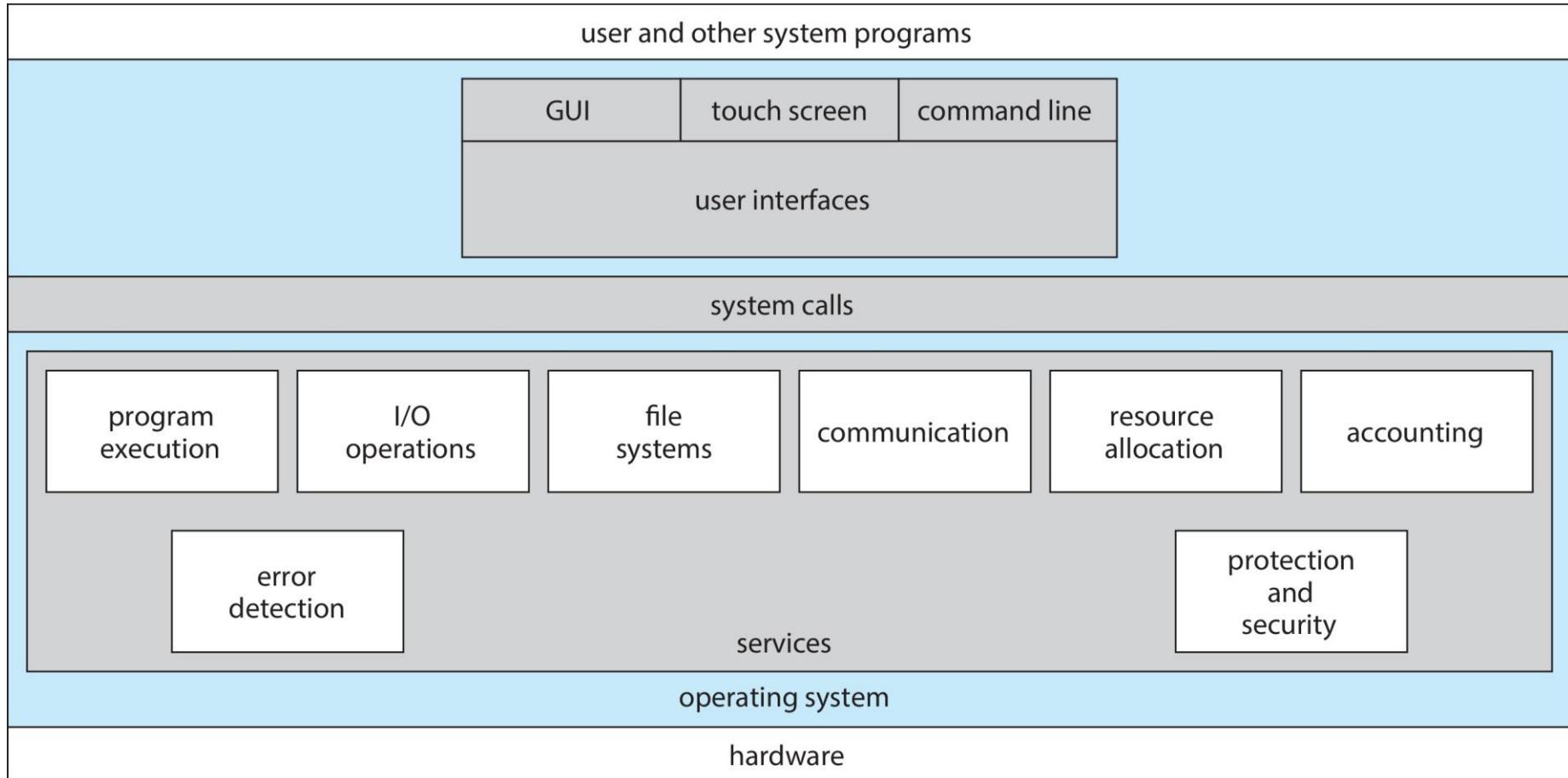


눈높이 체크

- 운영 체제 서비스를 알고 계신가요?
- 사용자 및 운영 체제 인터페이스를 알고 계신가요?
- 시스템 호출을 알고 계신가요?
- 시스템 서비스를 알고 계신가요?
- 링커 및 로더를 알고 계신가요?
- 응용 프로그램이 운영 체제에 특정한 이유를 알고 계신가요?
- 설계 및 구현을 알고 계신가요?
- 운영 체제 구조를 알고 계신가요?
- 운영 체제 구축 및 부팅을 알고 계신가요?
- 운영 체제 디버깅을 알고 계신가요?

1. Operating System Services

운영 체제 서비스 보기





1. Operating System Services

운영 체제 서비스 보기

- 운영 체제는 프로그램과 사용자에게 프로그램과 서비스를 실행하기 위한 환경을 제공.
- 한 세트의 운영 체제 서비스는 사용자에게 유용한 기능을 제공.
 - 사용자 인터페이스 - 거의 모든 운영 체제에는 사용자 인터페이스(UI)가 있다.
 - 명령줄(CLI), 그래픽 사용자 인터페이스(GUI), 터치스크린, Batch에 따라 다름
 - 프로그램 실행 - 시스템은 프로그램을 메모리에 로드하고 해당 프로그램을 실행하고 정상 또는 비정상(오류 표시)으로 실행을 종료할 수 있다.
 - I/O 작업 - 실행 중인 프로그램에는 파일 또는 I/O 장치와 관련된 I/O가 필요할 수 있다.
 - 파일 시스템 조작 - 파일 시스템은 특히 중요. 프로그램은 파일과 디렉토리를 읽고 쓰고, 만들고 삭제하고, 검색하고, 파일 정보를 나열하고, 권한을 관리.



1. Operating System Services

운영 체제 서비스 보기

- 한 세트의 운영 체제 서비스는 사용자에게 유용한 기능을 제공.
 - 통신 – 프로세스는 동일한 컴퓨터에서 또는 네트워크를 통해 컴퓨터 간에 정보를 교환할 수 있다.
 - ✓ 통신은 공유 메모리 또는 메시지 전달(OS에 의해 이동된 패킷)을 통해 이루어질 수 있다.
 - 오류 감지 – OS는 가능한 오류를 지속적으로 인식.
 - ① CPU 및 메모리 하드웨어, I/O 장치, 사용자 프로그램에서 발생할 수 있다.
 - ② 각 유형의 오류에 대해 OS는 정확하고 일관된 컴퓨팅을 보장하기 위해 적절한 조치를 취해야 한다.
 - ③ 디버깅 기능은 시스템을 효율적으로 사용하는 사용자와 프로그래머의 능력을 크게 향상시킬 수 있다.



1. Operating System Services

운영 체제 서비스 보기

- 리소스 공유를 통해 시스템 자체의 효율적인 운영을 보장하기 위한 또 다른 OS 기능 세트가 존재.
- 리소스 할당 - 여러 사용자 또는 여러 작업이 동시에 실행되는 경우 각 사용자에게 리소스를 할당.
- ✓ 많은 유형의 리소스 - CPU 주기, 주 메모리, 파일 저장소, I/O 장치.
- 로깅 - 어떤 사용자가 컴퓨터 리소스를 얼마나, 어떤 종류로 사용하는지 추적하기 위해
- 보호 및 보안 Protection and security - 다중 사용자 또는 네트워크 컴퓨터 시스템에 저장된 정보의 소유자는 해당 정보의 사용을 제어하기를 원할 수 있으며 동시 프로세스는 서로 간섭해서는 안 됨.
- 보호 Protection에는 시스템 리소스에 대한 모든 액세스를 제어하는 것이 포함.
- 보안 Security 외부인으로부터 시스템을 보호하려면 사용자 인증이 필요하며 유효하지 않은 액세스 시도로부터 외부 I/O 장치를 보호하는 것으로 확장.



2. User and Operating System-Interface

Command Line interpreter

- CLI는 직접 명령 입력을 허용.
- 때로는 커널에서, 때로는 시스템 프로그램에 의해 구현됨
- 때때로 여러 가지 맛이 구현됨 – 쉘shells
- 주로 사용자로부터 명령을 가져와 실행.
- 때로는 명령이 내장되어 있고 때로는 프로그램 이름만 있다.
- 후자의 경우 새 기능을 추가할 때 셀 수정이 필요하지 않다.



2. User and Operating System-Interface

User Operating System Interface - GUI

- 사용자 친화적인 데스크탑 메타포 인터페이스
 - 일반적으로 마우스, 키보드 및 모니터
 - 아이콘은 파일, 프로그램, 작업 등.
 - 인터페이스의 개체에 대한 다양한 마우스 버튼은 다양한 동작(정보 제공, 옵션 제공, 기능 실행, 디렉토리(폴더라고 함) 열기)을 유발.
 - Xerox PARC에서 발명
- 이제 많은 시스템에 CLI 및 GUI 인터페이스가 모두 포함.
 - Microsoft Windows는 CLI "명령" 셸이 있는 GUI.
 - Apple Mac OS X는 Unix 커널과 사용 가능한 셸이 있는 "Aqua" GUI 인터페이스.
 - Unix 및 Linux에는 선택적 GUI 인터페이스(CDE, KDE, GNOME)가 있는 CLI가 있다.

2. User and Operating System-Interface



Touchscreen Interfaces

- 터치스크린 장치에는 새로운 인터페이스가 필요.
 - 마우스를 사용할 수 없거나 원하지 않음
 - 제스처에 따른 동작 및 선택
 - 텍스트 입력을 위한 가상 키보드
 - 음성 명령





3. System Calls

System Calls

- OS에서 제공하는 서비스에 대한 프로그래밍 인터페이스
- 일반적으로 고급 언어(C 또는 C++)로 작성됨
- 직접적인 시스템 호출 사용보다는 상위 수준 API(Application Programming Interface)를 통해 프로그램에서 대부분 액세스
- 세 가지 가장 일반적인 API는 Windows용 Win32 API, POSIX 기반 시스템용 POSIX API(UNIX, Linux 및 Mac OS X의 거의 모든 버전 포함) 및 JVM(Java Virtual Machine)용 Java API.

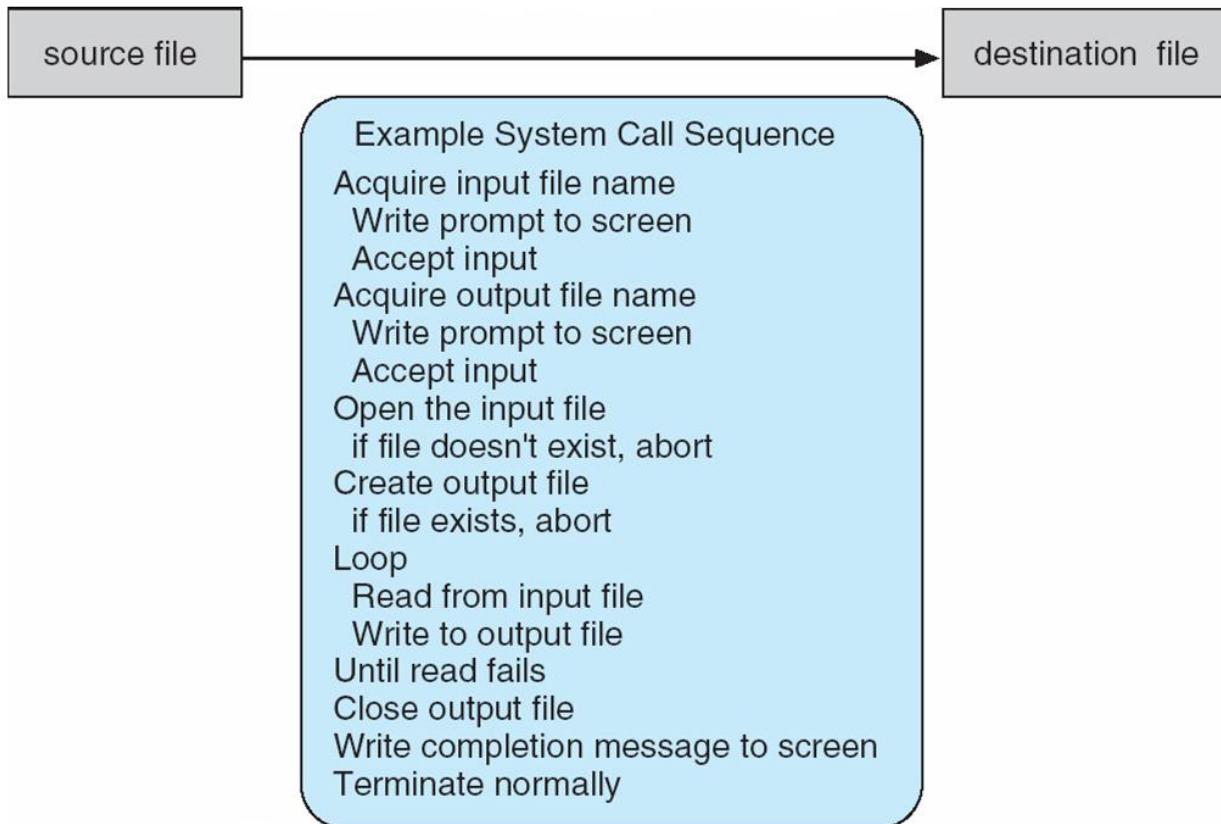


3. System Calls

System Calls

- Example of System Calls

- 한 파일의 내용을 다른 파일로 복사하는 시스템 호출 시퀀스





3. System Calls

System Calls

- Example of Standard API

```
#include <unistd.h>

ssize_t      read(int fd, void *buf, size_t count)
```

return
value

function
name

parameters



3. System Calls

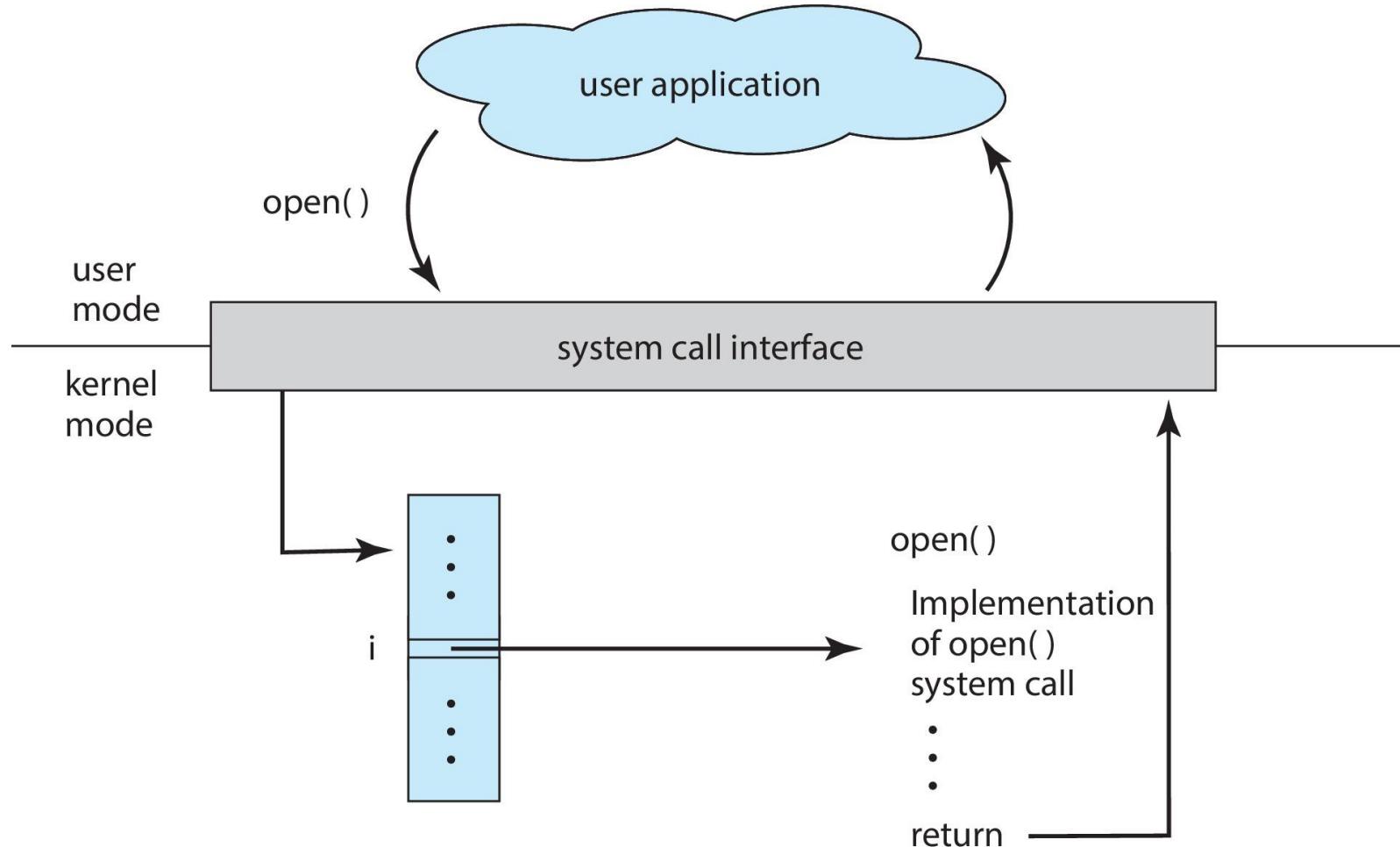
System Call Implementation

- 일반적으로 번호는 각 시스템 호출과 연결.
 - 시스템 호출 인터페이스는 이러한 번호에 따라 색인이 생성된 테이블을 유지.
- 시스템 호출 인터페이스는 OS 커널에서 의도한 시스템 호출을 호출하고 시스템 호출의 상태와 모든 반환 값을 반환.
- 호출자는 시스템 호출이 어떻게 구현되는지에 대해 알 필요가 없다.
 - API를 준수하고 OS가 결과 호출로 수행할 작업을 이해하기만 하면 됨.
 - API에 의해 프로그래머에게 숨겨진 OS 인터페이스의 대부분의 세부 정보 런타임 지원 라이브러리로 관리(컴파일러에 포함된 라이브러리에 내장된 함수 집합)

3. System Calls

System Call Implementation

- API – System Call – OS Relationship





3. System Calls

시스템 호출 매개변수 전달

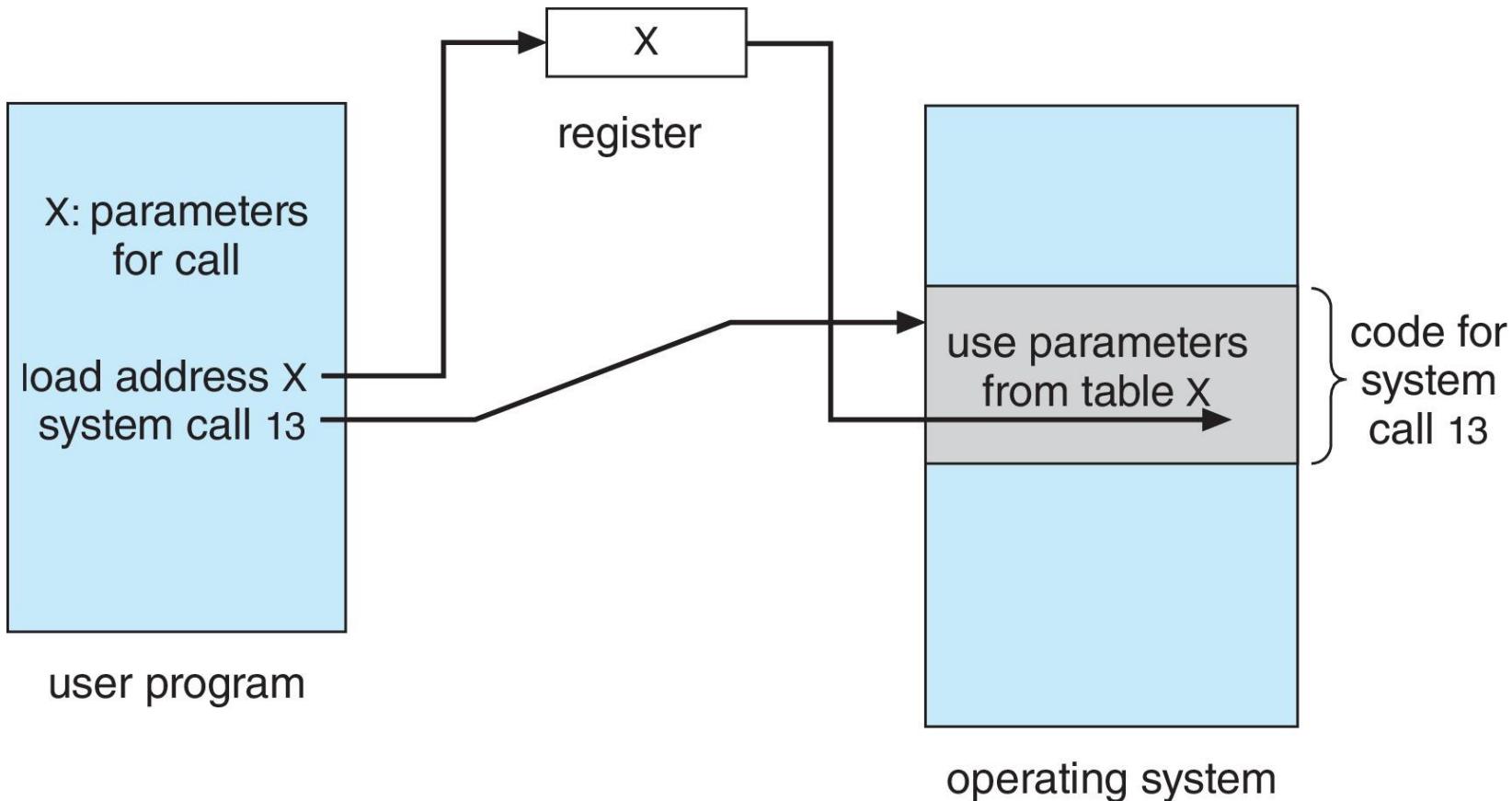
- 종종 원하는 시스템 호출의 ID보다 더 많은 정보가 필요.
 - 정확한 정보의 종류와 양은 OS와 호출에 따라 다름
- OS에 매개변수를 전달하는 데 사용되는 세 가지 일반적인 방법
 - 1) 가장 간단함: 레지스터에 매개변수 전달
 - 경우에 따라 레지스터보다 매개변수가 더 많을 수 있다.
 - 2) 메모리의 블록 또는 테이블에 저장된 매개변수 및 레지스터에서 매개변수로 전달된 블록의 주소
 - Linux 및 Solaris에서 채택한 이 접근 방식
 - 3) 프로그램에 의해 스택에 배치되거나 푸시되고 운영 체제에 의해 스택에서 꺼지는 매개변수
 - 4) 블록 및 스택 방법은 전달되는 매개변수의 수 또는 길이를 제한하지 않는다.



3. System Calls

시스템 호출 매개변수 전달

- 테이블을 통한 매개변수 전달





3. System Calls

Types of System Calls

- **프로세스 제어**

- 프로세스 생성, 프로세스 종료
- 끝내다, 중단하다
- 로드, 실행
- 프로세스 속성 가져오기, 프로세스 속성 설정
- 시간을 기다리다
- 대기 이벤트, 신호 이벤트
- 메모리 할당 및 해제
- 오류가 발생하면 메모리 덤프
- 버그 판별을 위한 디버거 Debugger, 단일 단계 실행
- 프로세스 간 공유 데이터에 대한 액세스 관리를 위한 잠금 **Locks**



3. System Calls

Types of System Calls

- 파일 관리

- 파일 생성, 파일 삭제
- 파일 열기, 닫기
- 읽기, 쓰기, 재배치
- 파일 속성 가져오기 및 설정

- 장치 관리

- 요청 장치, 릴리스 장치
- 읽기, 쓰기, 재배치
- 장치 속성 가져오기, 장치 속성 설정
- 장치를 논리적으로 연결 또는 분리



3. System Calls

Types of System Calls

- 정보 유지

- 시간 또는 날짜 가져오기, 시간 또는 날짜 설정
- 시스템 데이터 가져오기, 시스템 데이터 설정
- 프로세스, 파일 또는 장치 속성 가져오기 및 설정

- Communications

- 통신 연결 생성, 삭제
- 메시지 전달 모델 message passing model 이 호스트 이름 또는 프로세스 이름인 경우 메시지 보내기, 받기
- ✓ 클라이언트에서 서버로
- 공유 메모리 모델 Shared-memory model 생성 및 메모리 영역 액세스 권한 획득
- 전송 상태 정보
- 원격 장치 연결 및 분리



3. System Calls

Types of System Calls

- 보호

- 리소스에 대한 액세스 제어
- 권한 가져오기 및 설정
- 사용자 액세스 허용 및 거부



3. System Calls

Examples of System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

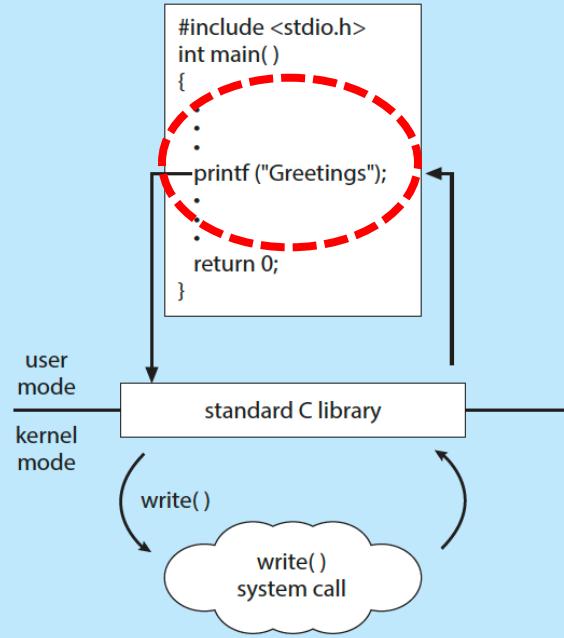
The following illustrates various equivalent system calls for Windows and UNIX operating systems.



	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:





3. System Calls

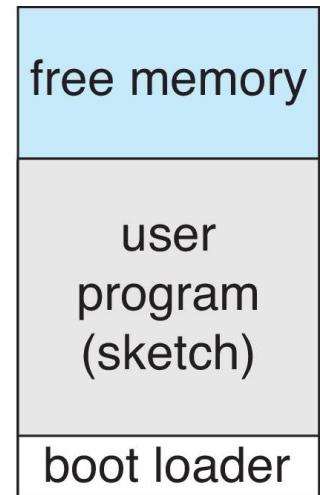
Example: Arduino

- 단일 작업
- 운영 체제 없음
- USB를 통해 플래시 메모리에 로드된 프로그램(스케치)
- 단일 메모리 공간
- 부트 로더가 프로그램을 로드.
- 프로그램 종료 -> 웰 다시 로드됨



(a)

At system startup



(b)

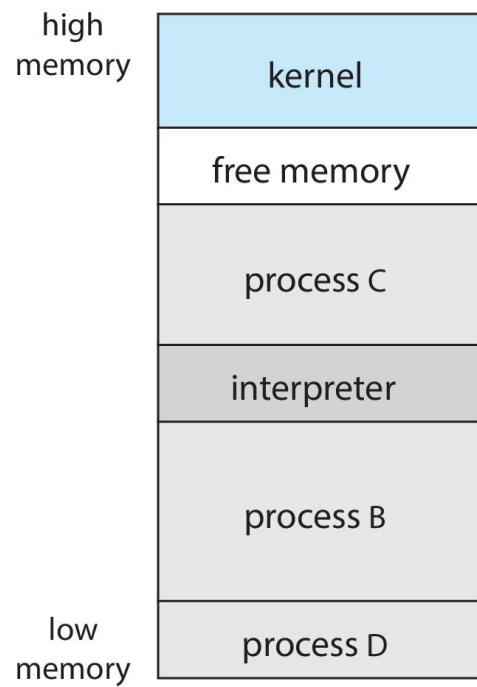
running a program



3. System Calls

Example: FreeBSD

- 유닉스 변종
- 멀티태스킹
- 사용자 로그인 -> 사용자가 선택한 쉘 호출
- Shell은 fork() 시스템 호출을 실행하여 프로세스를 생성.
 - exec()를 실행하여 프로그램을 프로세스에 로드.
 - 쉘은 프로세스가 종료될 때까지 기다리거나 사용자 명령을 계속 사용.
- 프로세스 종료:
 - 코드 = 0 – 오류 없음
 - 코드 > 0 – 오류 코드





4. System Services

System Services?

- 시스템 프로그램은 프로그램 개발 및 실행에 편리한 환경을 제공. 다음과 같이 나눌 수 있다.
 - 파일 조작
 - 때때로 파일에 저장되는 상태 정보
 - 프로그래밍 언어 지원
 - 프로그램 로딩 및 실행
 - 연락
 - 백그라운드 서비스
 - 응용 프로그램
- 운영 체제에 대한 대부분의 사용자 관점은 실제 시스템 호출이 아니라 시스템 프로그램에 의해 정의.



4. System Services

System Services?

- 프로그램 개발 및 실행에 편리한 환경 제공
 - 그들 중 일부는 단순히 시스템 호출에 대한 사용자 인터페이스. 다른 것들은 훨씬 더 복잡
- 파일 관리 - 파일 및 디렉토리 생성, 삭제, 복사, 이름 바꾸기, 인쇄, 덤프, 나열 및 일반적으로 조작
- 상태 정보
 - 일부는 시스템에 날짜, 시간, 사용 가능한 메모리 양, 디스크 공간, 사용자 수와 같은 정보를 요청.
 - 기타 자세한 성능, 로깅 및 디버깅 정보를 제공.
 - 일반적으로 이러한 프로그램은 형식을 지정하고 출력을 터미널이나 기타 출력 장치로 인쇄.
 - 일부 시스템은 구성 정보를 저장하고 검색하는 데 사용되는 레지스트리를 구현.



4. System Services

System Services?

- 파일 수정
 - 파일 생성 및 수정을 위한 텍스트 편집기
 - 파일 내용을 검색하거나 텍스트 변환을 수행하는 특수 명령
- 프로그래밍 언어 지원 - 때때로 제공되는 컴파일러, 어셈블러, 디버거 및 인터프리터
- 프로그램 로딩 및 실행 - 앱솔루트 로더, 재배치 가능 로더, 링키지 에디터, 오버레이 로더, 고급 및 기계어용 디버깅 시스템
- 통신 Communications - 프로세스, 사용자 및 컴퓨터 시스템 간에 가상 연결을 생성하는 메커니즘을 제공.
- 사용자가 서로의 화면에 메시지를 보내고, 웹 페이지를 탐색하고, 전자 메일 메시지를 보내고, 원격으로 로그인하고, 한 시스템에서 다른 시스템으로 파일을 전송할 수 있다.



4. System Services

System Services?

- 백그라운드 서비스

- 부팅 시 실행
 - ✓ 일부는 시스템 시작 후 종료
 - ✓ 일부는 시스템 부팅에서 종료까지
- 디스크 검사, 프로세스 스케줄링, 오류 로깅, 인쇄와 같은 기능 제공
- 커널 컨텍스트가 아닌 사용자 컨텍스트에서 실행
- 서비스, 하위 시스템, 데몬으로 알려져 있음

- 응용 프로그램

- 시스템과 관련 없음
- 사용자가 실행
- 일반적으로 OS의 일부로 간주되지 않음
- 명령줄, 마우스 클릭, 손가락 찌르기^{finger poke}로 실행



5. Linkers and Loaders

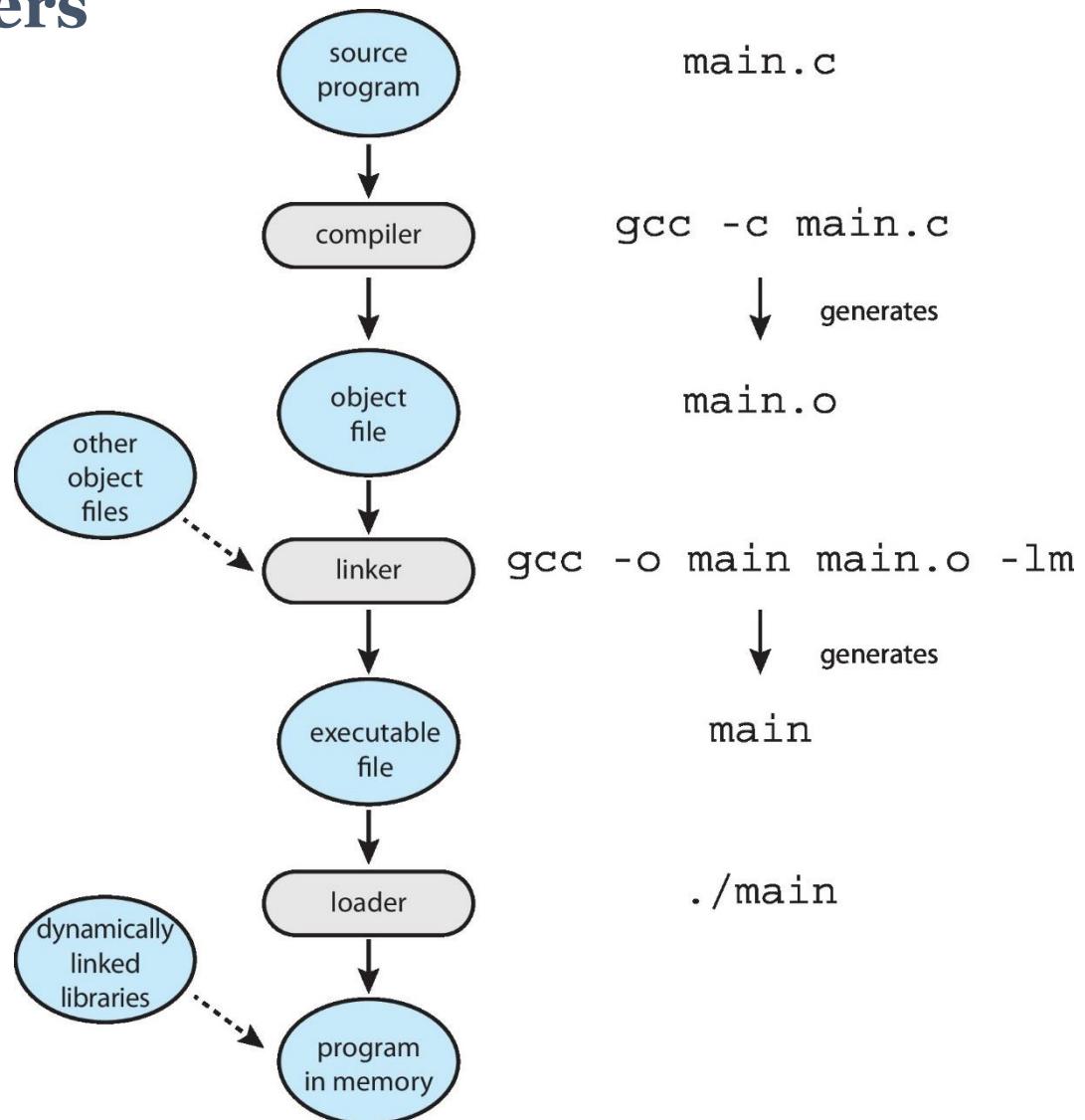
Linkers and Loaders

- 의의 물리적 메모리 위치에 로드되도록 설계된 개체 파일로 컴파일 된 소스 코드 – 재배치 가능한 개체 파일
- 링커는 이들을 단일 바이너리 실행 파일로 결합.
 - 또한 라이브러리를 가져옴.
- 프로그램은 바이너리 실행 파일로 보조 스토리지에 상주.
- 실행될 로더 loader에 의해 메모리로 가져와야 함
 - 재배치 Relocation 는 프로그램 부분에 최종 주소를 할당하고 해당 주소 와 일치하도록 프로그램의 코드와 데이터를 조정.
- 최신 범용 시스템은 라이브러리를 실행 파일에 연결하지 않는다.
 - 오히려 동적으로 연결된 라이브러리(Windows에서는 DLL)가 필요에 따라 로드되고 동일한 라이브러리의 동일한 버전을 사용하는 모든 사용자가 공유(한 번 로드됨).
- 개체, 실행 파일에는 표준 형식이 있으므로 운영 체제에서 파일을 로드하고 시작하는 방법을 알고 있다.

5. Linkers and Loaders

Linkers and Loaders

- 링커와 로더의 역할





5. Linkers and Loaders

Linkers and Loaders

파일

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/바탕화면$ cd ~
```

```
k8s@DESKTOP-RoEQ2U6:~$ ls
```

공개 다운로드 문서 바탕화면 비디오 사진 음악 템플릿

```
k8s@DESKTOP-RoEQ2U6:~$ mkdir c-workspaces
```

```
k8s@DESKTOP-RoEQ2U6:~$ ls
```

c-workspaces 공개 다운로드 문서 바탕화면 비디오 사진 음악 템플릿

```
k8s@DESKTOP-RoEQ2U6:~$ cd c-workspaces/
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ls
```

실습환경

소스코드

결과값1

비고



5. Linkers and Loaders

Linkers and Loaders

- 암시적 실행 파일이름

파일	<u>소스코드</u>
실습환경	<pre>k8s@DESKTOP-RoEQ2U6:~/c_workspaces\$ cat > helloworld.c #include <stdio.h> int main(int argc, char *argv[]){ printf("HELLO WORLD !! \n"); return 0; }</pre>
소스코드	<pre>k8s@DESKTOP-RoEQ2U6:~/c_workspaces\$ gcc helloworld.c k8s@DESKTOP-RoEQ2U6:~/c_workspaces\$./a.out HELLO WORLD !!</pre>
결과값1	
비고	

- 명시적 실행 파일이름

파일	<u>소스코드</u>
실습환경	
소스코드	
결과값1	<pre>k8s@DESKTOP-RoEQ2U6:~/c_workspaces\$ gcc -o hello helloworld.c k8s@DESKTOP-RoEQ2U6:~/c_workspaces\$./hello HELLO WORLD !! k8s@DESKTOP-RoEQ2U6:~/c_workspaces\$</pre>
비고	



5. Linkers and Loaders

Linkers and Loaders

- 암시적 실행 파일이름

파일

소스코드

실습환경

소스코드

Sample.h

```
#include <stdio.h>
void fun1();
void fun2();
```

main.c

```
#include "Sample.h"
int WinMain(){
    fun1();
    fun2();
    return 0;
}
```

fun1.c

```
#include <stdio.h>
void fun1(){
    printf("fun1 !! \n");
}
```

fun2.c

```
#include <stdio.h>
void fun2(){
    printf("fun2 !! \n");
}
```



5. Linkers and Loaders

Linkers and Loaders

● 암시적 실행 파일이름

파일

소스코드

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ gcc -c fun1.c
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ gcc -c fun2.c
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ gcc -c main.c
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ ls -al
....
-rw-r--r-- 1 k8s k8s 45 May 14 14:55 Sample.h
-rw-r--r-- 1 k8s k8s 57 May 14 14:58 fun1.c
-rw-r--r-- 1 k8s k8s 1488 May 14 14:58 fun1.o
-rw-r--r-- 1 k8s k8s 58 May 14 14:58 fun2.c
-rw-r--r-- 1 k8s k8s 1488 May 14 14:58 fun2.o
-rw-r--r-- 1 k8s k8s 63 May 14 15:03 main.c
-rw-r--r-- 1 k8s k8s 1416 May 14 15:03 main.o
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ gcc -o main main.o fun1.o fun2.o
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ main
fun1 !!
fun2 !!
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$
```

비고



6. Applications

응용 프로그램이 운영 체제에 특정한 이유

- 일반적으로 한 시스템에서 컴파일된 앱은 다른 운영 체제에서 실행 할 수 없다.
- 각 운영 체제는 고유한 시스템 호출을 제공.
 - 자체 파일 형식 등
- 앱은 다중 운영 체제일 수 있다.
 - 여러 운영 체제에서 사용 가능한 Python, Ruby 및 인터프리터와 같은 해석 언어로 작성
 - 실행 중인 앱을 포함하는 VM을 포함하는 언어로 작성된 앱(예: Java)
 - 표준 언어(예: C)를 사용하고 각 운영 체제에서 개별적으로 컴파일하여 각 운영 체제에서 실행
- ABI(Application Binary Interface)는 API와 동등한 아키텍처로, 주어진 아키텍처, CPU 등에서 주어진 운영 체제에 대해 서로 다른 바이너리 코드 구성 요소가 인터페이스할 수 있는 방법을 정의.



7. Design and Implementation

Design and Implementation

- OS의 설계 및 구현은 "해결 가능 solvable"하지 않지만 일부 접근 방식은 성공적인 것으로 입증되었다.
- 서로 다른 운영 체제의 내부 구조는 크게 다를 수 있다.
- 목표와 사양을 정의하여 설계 시작
- 하드웨어 선택, 시스템 유형의 영향을 받음
- 사용자 목표 및 시스템 목표
 - 사용자 목표 User goals – 운영 체제는 사용하기 편리하고, 배우기 쉽고, 안정적이고, 안전하고, 빨라야 한다.
 - 시스템 목표 System goals – 운영 체제는 유연하고 안정적이며 오류가 없고 효율적일 뿐만 아니라 설계, 구현 및 유지 관리가 쉬워야 한다.
- OS를 지정하고 설계하는 것은 소프트웨어 엔지니어링의 매우 창의적인 작업.



7. Design and Implementation

Policy and Mechanism

- 정책 Policy : 무엇을 해야 합니까?
 - 예: 100초마다 인터럽트
- 메커니즘 Mechanism : 어떤 작업을 수행하는 방법?
 - 예: 타이머
- 중요 원칙: 메커니즘과 정책 분리
- 메커니즘에서 정책을 분리하는 것은 매우 중요한 원칙이며 정책 결정이 나중에 변경될 경우 최대한의 유연성을 허용.
 - 예: 100을 200으로 변경



7. Design and Implementation

Implementation

- 많은 변형
 - 어셈블리 언어의 초기 OS
 - 그런 다음 Algol, PL/1과 같은 시스템 프로그래밍 언어
 - 이제 C, C++
- 실제로 일반적으로 언어의 혼합
 - 어셈블리에서 가장 낮은 수준
 - C의 본체
 - C, C++의 시스템 프로그램, PERL, Python, 웰 스크립트와 같은 스크립팅 언어
- 다른 하드웨어로 포팅하기 쉬운 고급 언어
 - 그러나 느리게
- 에뮬레이션은 OS가 네이티브가 아닌 하드웨어에서 실행되도록 허용할 수 있다.



8. Operating System Structure

General-purpose OS

- 범용 OS는 매우 큰 프로그램.
- 구성하는 다양한 방법
 - 간단한 구조 – MS-DOS
 - 더 복잡한 – 유닉스
 - 계층화 – 추상화 an abstraction
 - 마이크로커널 – 마하 Mach



8. Operating System Structure

일체형 구조- Monolithic Structure

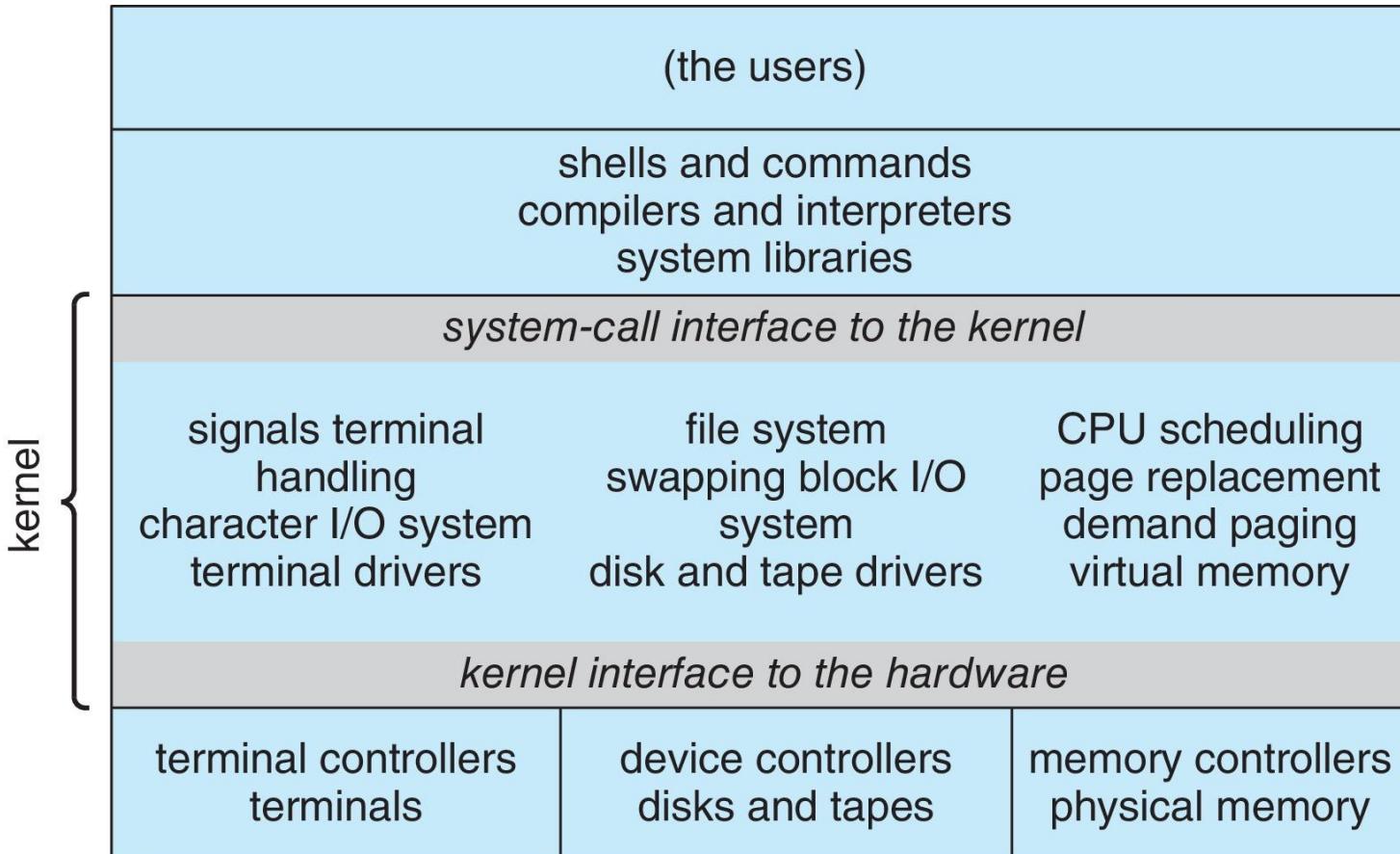
- **Monolithic Structure – Original UNIX**
- **UNIX** – 하드웨어 기능에 의해 제한되는 원래 UNIX 운영 체제는 구조가 제한적이었다.
- UNIX OS는 분리 가능한 두 부분으로 구성.
 - 시스템 프로그램
 - 커널
 - ✓ 시스템 호출 인터페이스 아래와 물리적 하드웨어 위의 모든 것으로 구성
 - ✓ 파일 시스템, CPU 스케줄링, 메모리 관리 및 기타 운영 체제 기능을 제공합니다. 한 수준에 대한 많은 기능



8. Operating System Structure

Traditional UNIX System Structure

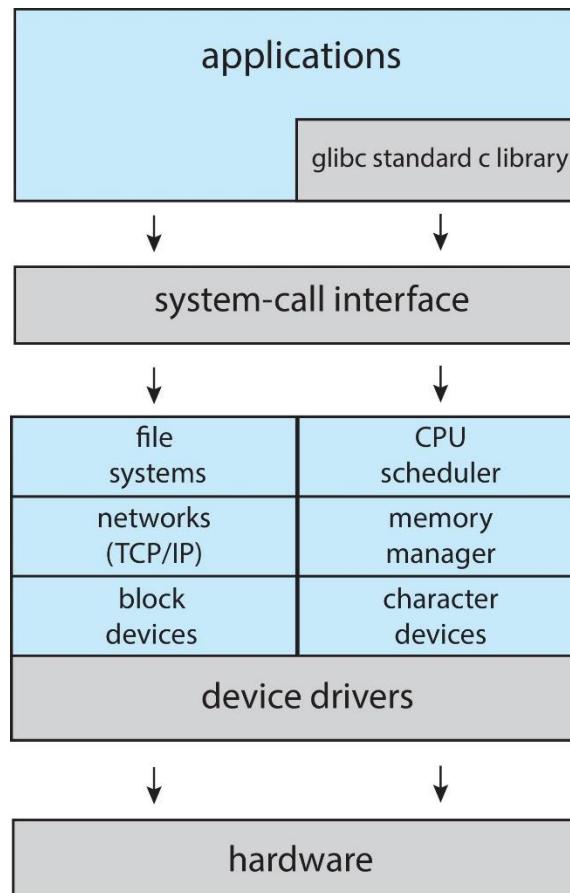
- 단순하지만 완전히 계층화되지 않은 것 이상



8. Operating System Structure

Linux System Structure

- 모듈리식 플러스 모듈식 설계

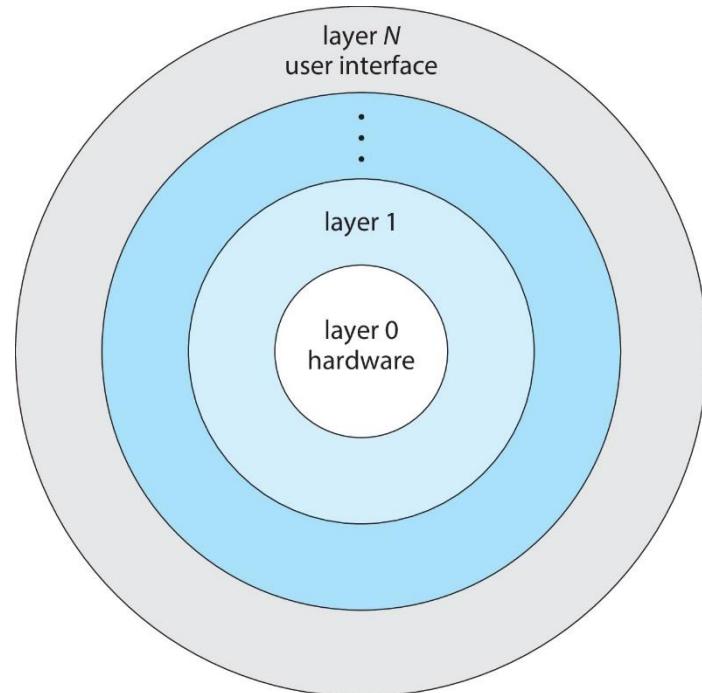




8. Operating System Structure

계층적 접근

- 운영 체제는 여러 계층(수준)으로 나뉘며 각 계층은 하위 계층 위에 구축됩니다. 맨 아래 계층(계층 0)은 하드웨어. 최상위(계층 N)는 사용자 인터페이스.
- 모듈화를 통해 각 레이어는 하위 레이어의 기능(작업) 및 서비스만 사용하도록 레이어를 선택.





8. Operating System Structure

Microkernels

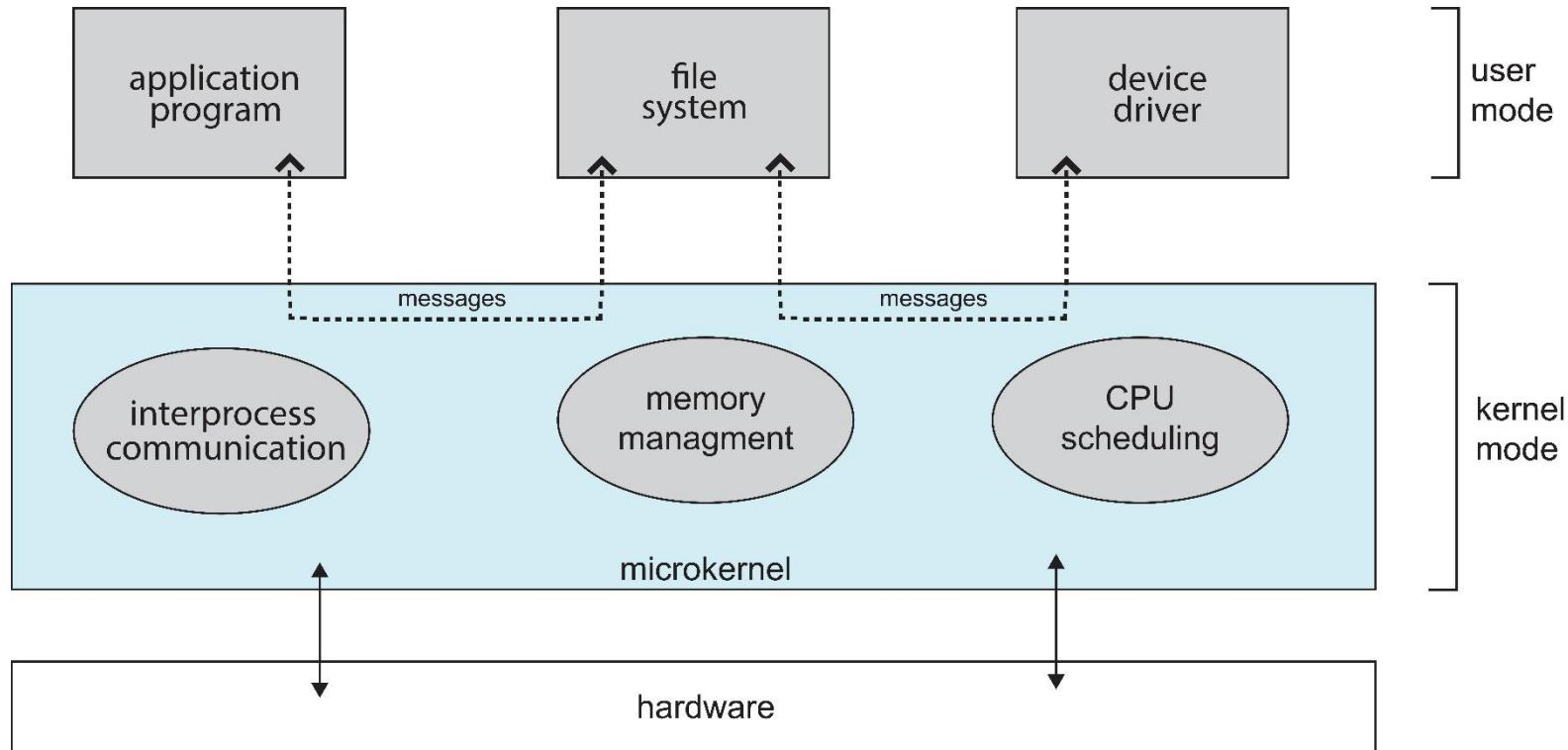
- 커널에서 사용자 공간으로 이동
- Mach는 마이크로 커널의 예.
- 부분적으로 Mach를 기반으로 하는 Mac OS X 커널(Darwin)
- 메시지 전달을 사용하여 사용자 모듈 간에 통신이 이루어진다.
- 이익 Benefits :
 - 마이크로커널 확장 용이
 - 운영 체제를 새로운 아키텍처로 이식하기가 더 쉽다.
 - 안정성 향상(커널 모드에서 실행되는 코드가 적음)
 - 더 안전한
- 손해 Detriments :
 - 커널 공간 통신에 대한 사용자 공간의 성능 오버헤드



8. Operating System Structure

Microkernels

- Microkernel System Structure





8. Operating System Structure

Modules

- 많은 최신 운영 체제는 로드 가능한 커널 모듈 **loadable kernel modules**(LKM)을 구현.
 - 객체 지향 접근 방식을 사용.
 - 각 핵심 구성 요소는 별개.
 - 각각은 알려진 인터페이스를 통해 서로 대화.
 - 각각은 커널 내에서 필요에 따라 로드할 수 있다.
- 전반적으로 레이어와 비슷하지만 더 유연.
 - 리눅스, 솔라리스 등



8. Operating System Structure

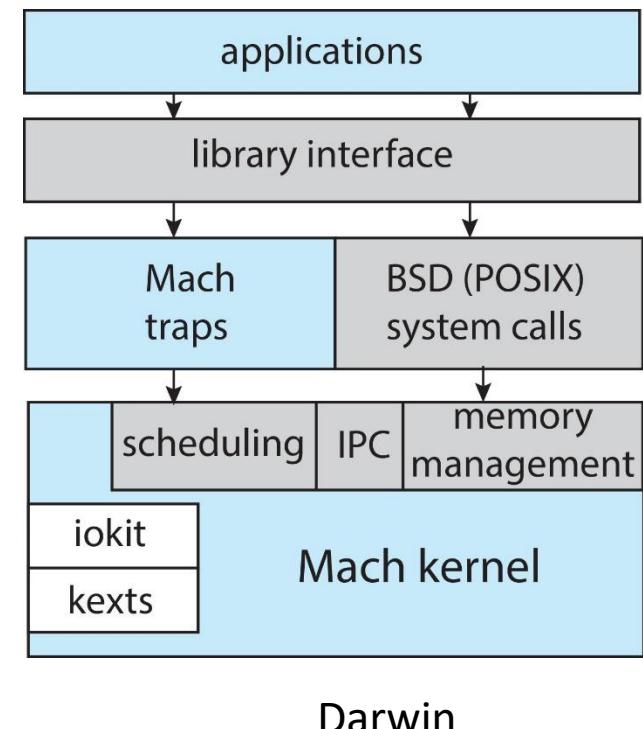
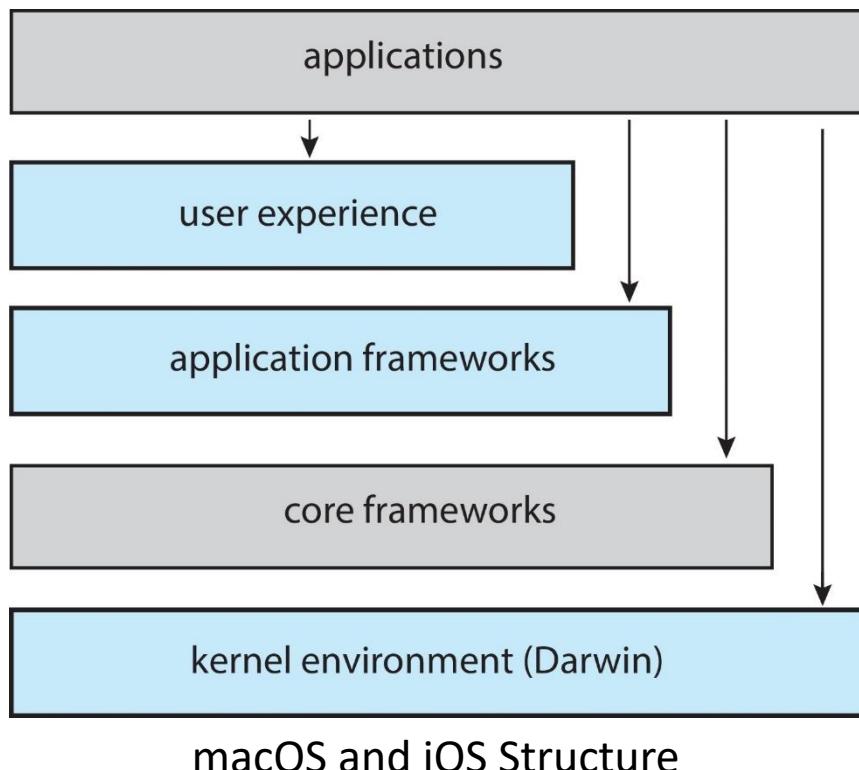
Hybrid Systems

- 대부분의 최신 운영 체제는 하나의 순수한 모델이 아니다.
 - 하이브리드는 성능, 보안, 유용성 요구 사항을 해결하기 위해 여러 접근 방식을 결합한다.
 - 커널 주소 공간의 Linux 및 Solaris 커널, 모듈리식, 기능의 동적 로드를 위한 모듈식 추가
 - Windows는 대부분 모듈리식이며 다양한 하위 시스템 특성을 위한 마이크로커널

8. Operating System Structure

Hybrid Systems

- Apple Mac OS X 하이브리드, 레이어드, Aqua UI 및 Cocoa 프로그래밍 환경
- 아래는 Mach 마이크로커널 및 BSD Unix 부품과 I/O 키트 및 동적으로 로드 가능한 모듈(커널 확장이라고 함)으로 구성된 커널.





8. Operating System Structure

iOS

- iPhone, iPad용 Apple 모바일 OS

- Mac OS X에 구조화, 기능 추가
- 기본적으로 OS X 응용 프로그램을 실행하지 않는다
- 또한 다른 CPU 아키텍처에서 실행됩니다(ARM 대 Intel).
- 앱 개발을 위한 Cocoa Touch Objective-C API
- 그래픽, 오디오, 비디오용 미디어 서비스 Media services 계층
- 핵심 서비스 Core services 는 클라우드 컴퓨팅, 데이터베이스를 제공.
- 핵심 운영 체제, Mac OS X 커널 기반

Cocoa Touch

Media Services

Core Services

Core OS



8. Operating System Structure

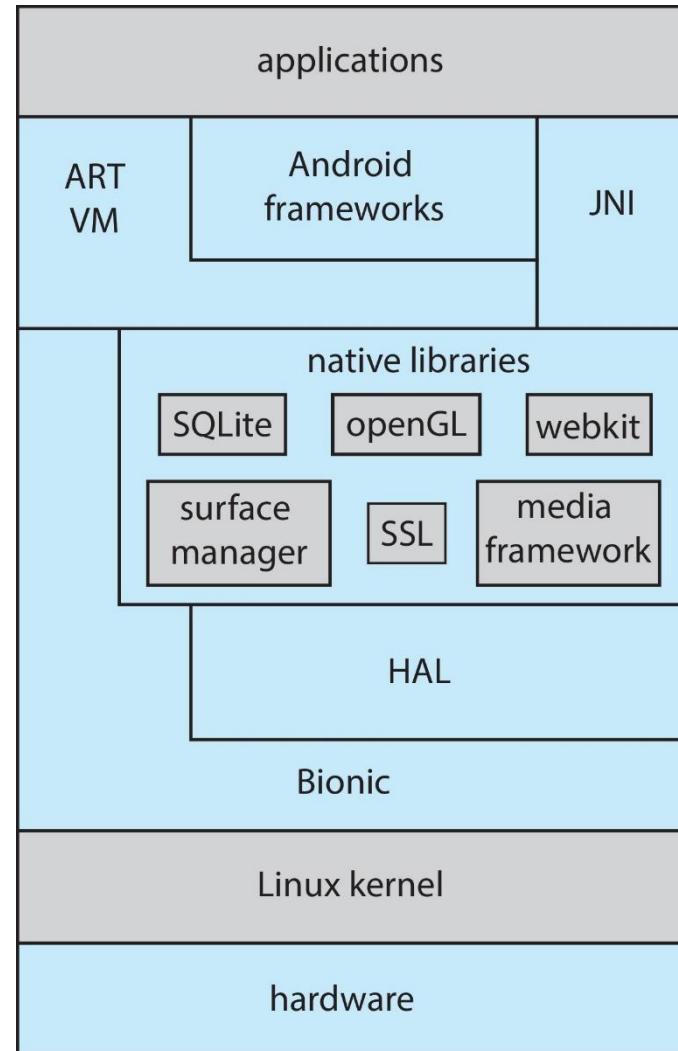
Android

- Open Handset Alliance(주로 Google)에서 개발
 - 오픈 소스
- iOS와 유사한 스택
- Linux 커널을 기반으로 하지만 수정됨
 - 프로세스, 메모리, 장치 드라이버 관리 제공
 - 전원 관리 추가
- 런타임 환경에는 핵심 라이브러리 세트와 Dalvik 가상 머신이 포함.
 - Java 및 Android API로 개발된 앱
 - Java 바이트코드로 컴파일된 후 실행 가능한 dex으로 변환된 Java 클래스 파일은 Dalvik VM에서 실행.
- 라이브러리에는 웹 브라우저(webkit), 데이터베이스(SQLite), 멀티미디어, 더 작은 libc용 프레임워크가 포함.

8. Operating System Structure

Android

● Android Architecture





9. 운영 체제 구축 및 부팅

운영 체제 구축 및 부팅

- 일반적으로 다양한 주변 장치가 있는 시스템 클래스에서 실행되도록 설계된 운영 체제
- 일반적으로 구입한 컴퓨터에 이미 설치된 운영 체제
 - 그러나 다른 운영 체제를 빌드하고 설치할 수 있다.
 - 처음부터 운영 체제를 생성하는 경우
- ① 운영 체제 소스 코드 작성
- ② 실행될 시스템에 대한 운영 체제 구성
- ③ 운영 체제 컴파일
- ④ 운영 체제 설치
- ⑤ 컴퓨터와 새 운영 체제를 부팅.



9. 운영 체제 구축 및 부팅

Building and Booting Linux

- Linux 소스 코드 다운로드(<http://www.kernel.org>)
- "make menuconfig"를 통해 커널 구성
- "make"를 사용하여 커널을 컴파일.
 - 커널 이미지 vmlinuz 생성
 - "make module"을 통해 커널 모듈 컴파일
 - "make module_install"을 통해 vmlinuz에 커널 모듈 설치
 - "make install"을 통해 시스템에 새 커널 설치



9. 운영 체제 구축 및 부팅

System Boot

- 시스템에서 전원이 초기화되면 고정된 메모리 위치에서 실행이 시작.
- 하드웨어가 운영 체제를 시작할 수 있도록 하드웨어에서 운영 체제를 사용할 수 있어야 한다.
- 작은 코드 조각 – ROM 또는 EEPROM에 저장된 부트스트랩 로더 boots trap loader, BIOS가 커널을 찾아 메모리에 로드하고 시작.
- 때때로 디스크에서 부트스트랩 로더를 로드하는 ROM 코드에 의해 로드된 고정된 위치에서 부트 블록이 있는 2단계 프로세스 two-step process
- 최신 시스템은 BIOS를 UEFI(Unified Extensible Firmware Interface)로 대체.
- 공통 부트스트랩 로더인 GRUB를 사용하면 여러 디스크, 버전, 커널 옵션에서 커널을 선택할 수 있다.
- 커널 로드 및 시스템 실행
- 부트 로더는 종종 단일 사용자 모드와 같은 다양한 부팅 상태를 허용.



9. 운영 체제 구축 및 부팅

Operating-System Debugging

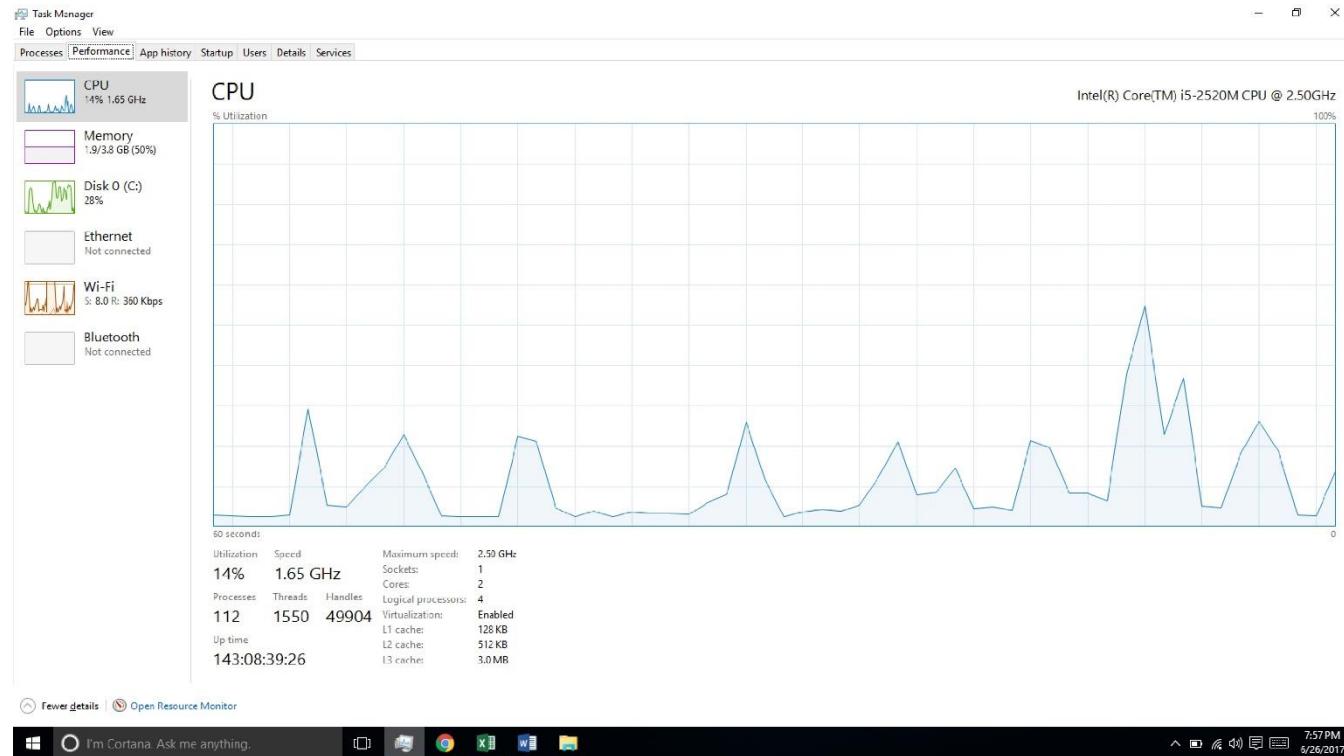
- 디버깅은 오류 또는 버그를 찾아 수정하는 것.
- 성능 튜닝
- OS는 오류 정보가 포함된 로그 파일을 생성.
- 응용 프로그램 오류는 프로세스의 메모리를 캡처하는 코어 덤프 파일을 생성할 수 있다.
- 운영 체제 오류로 인해 커널 메모리가 포함된 크래시 덤프 파일이 생성될 수 있음
- 충돌 외에도 성능 조정을 통해 시스템 성능을 최적화할 수 있다.
- 때때로 분석을 위해 기록된 활동의 추적 목록 *trace listings* 사용
- 프로파일링 **Profiling** 은 통계적 추세를 찾기 위한 명령 포인터의 주기적인 샘플링.
- Kernighan의 법칙: “디버깅은 처음부터 코드를 작성하는 것보다 두 배 더 어렵습니다.”



10. Performance Tuning

성능 조정

- 병목 현상을 제거하여 성능 향상
- OS는 시스템 동작을 계산하고 표시하는 수단을 제공.
- 예를 들어 "최상위" 프로그램 또는 Windows 작업 관리자





10. Performance Tuning

Tracing

- 시스템 호출 호출과 관련된 단계와 같은 특정 이벤트에 대한 데이터를 수집.
- 도구.
 - strace – 프로세스에 의해 호출된 추적 시스템 호출
 - gdb – 소스 레벨 디버거
 - perf – Linux 성능 도구 모음
 - tcpdump – 네트워크 패킷 수집



10. Performance Tuning

BCC

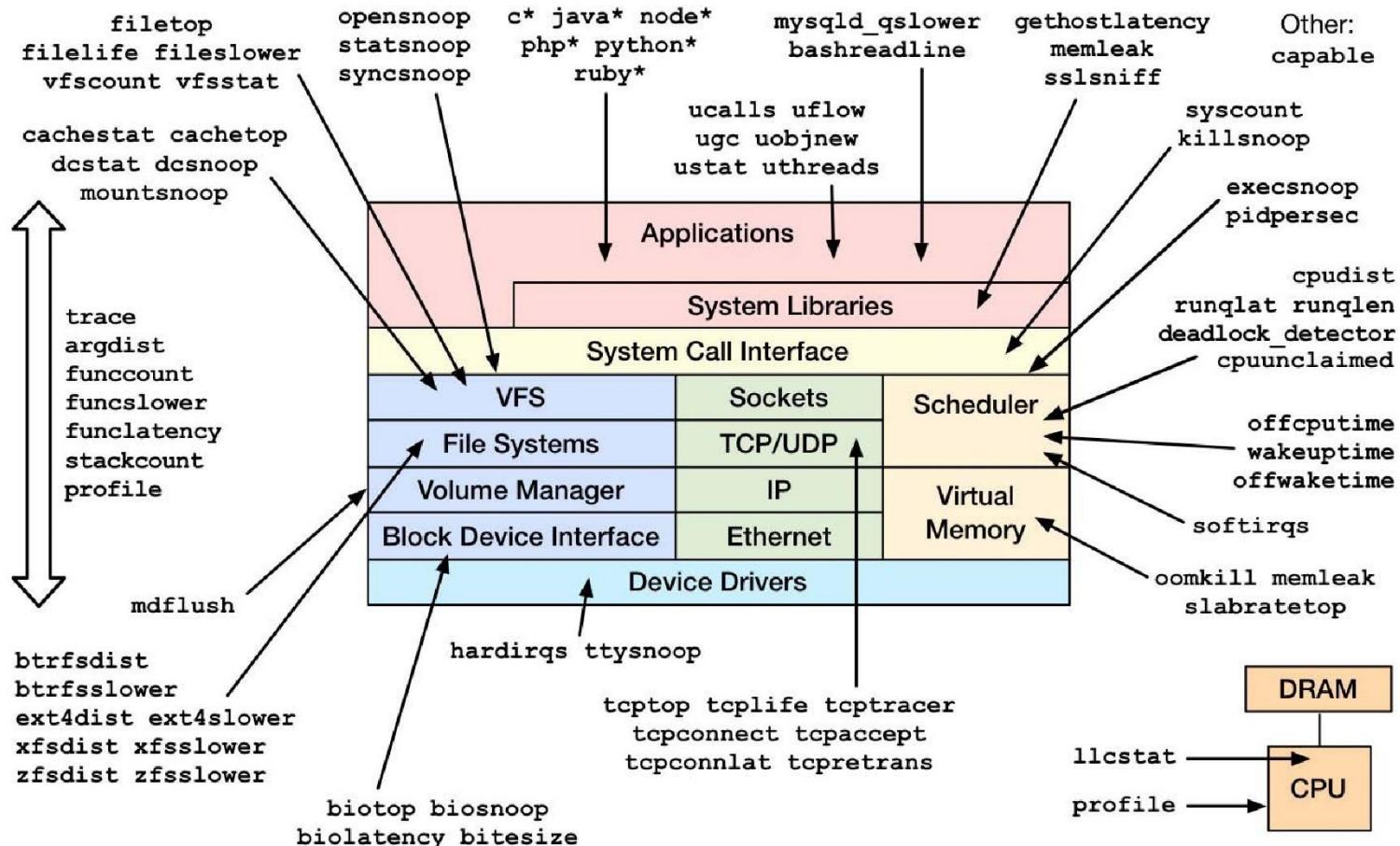
- 사용자 수준과 커널 코드 간의 상호 작용 디버깅은 둘 다 이해하고 해당 작업을 계측하는 도구 없이는 거의 불가능.
- BCC(BPF Compiler Collection)는 Linux용 추적 기능을 제공하는 풍부한 툴킷.
- 예를 들어 `disksnoop.py`는 디스크 I/O 활동을 추적.

TIME(s)	T	BYTES	LAT(ms)
1946.29186700	R	8	0.27
1946.33965000	R	8	0.26
1948.34585000	W	8192	0.96
1950.43251000	R	4096	0.56
1951.74121000	R	4096	0.35

10. Performance Tuning

Linux bcc/BPF Tracing Tools

Linux bcc/BPF Tracing Tools



『3과목』

6-8교시 :

Processes





Syllabus

학습목표

- 이 워크샵에서는 프로세스의 개별 구성 요소를 식별하고 운영 체제에서 해당 구성 요소가 어떻게 표시되고 예약되는지 설명할 수 있다.
- 이러한 작업을 수행하는 적절한 시스템 호출을 사용하여 프로그램을 개발하는 것을 포함하여 운영 체제에서 프로세스가 생성되고 종료되는 방법을 설명할 수 있다.
- 공유 메모리와 메시지 전달을 사용하는 프로세스 간 통신을 설명하고 대조할 수 있다.
- 파이프와 POSIX 공유 메모리를 사용하여 프로세스 간 통신을 수행하는 프로그램을 설계할 수 있다.
- 소켓과 원격 프로시저 호출을 사용한 클라이언트-서버 통신을 설명할 수 있다.
- Linux 운영 체제와 상호 작용하는 커널 모듈을 설계할 수 있다.



Syllabus

눈높이 체크

- 프로세스 개념을 알고 계신가요?
- 프로세스 스케줄링을 알고 계신가요?
- 프로세스에 대한 작업을 알고 계신가요?
- 프로세스 간 통신을 알고 계신가요?
- 공유 메모리 시스템의 IPC를 알고 계신가요?
- 메시지 전달 시스템의 IPC를 알고 계신가요?
- IPC 시스템의 예를 알고 계신가요?
- 클라이언트-서버 시스템의 통신을 알고 계신가요?



1. Process Concept

프로세스?

- 운영 체제는 프로세스로 실행되는 다양한 프로그램을 실행.
- 프로세스 **Process** – 실행 중인 프로그램. 프로세스 실행은 순차적으로 진행되어야 한다. 단일 프로세스의 명령을 병렬로 실행하지 않음
- 여러 부품
 - 텍스트 섹션 **text section** 이라고도 하는 프로그램 코드
 - 프로그램 카운터 **program counter**, 프로세서 레지스터를 포함한 현재 활동
 - 임시 데이터를 포함하는 스택 **Stack**
 - ✓ 함수 매개변수, 반환 주소, 지역 변수
 - 전역 변수를 포함하는 데이터 섹션 **Data section**
 - 런타임 동안 동적으로 할당된 메모리를 포함하는 힙 **Heap**



1. Process Concept

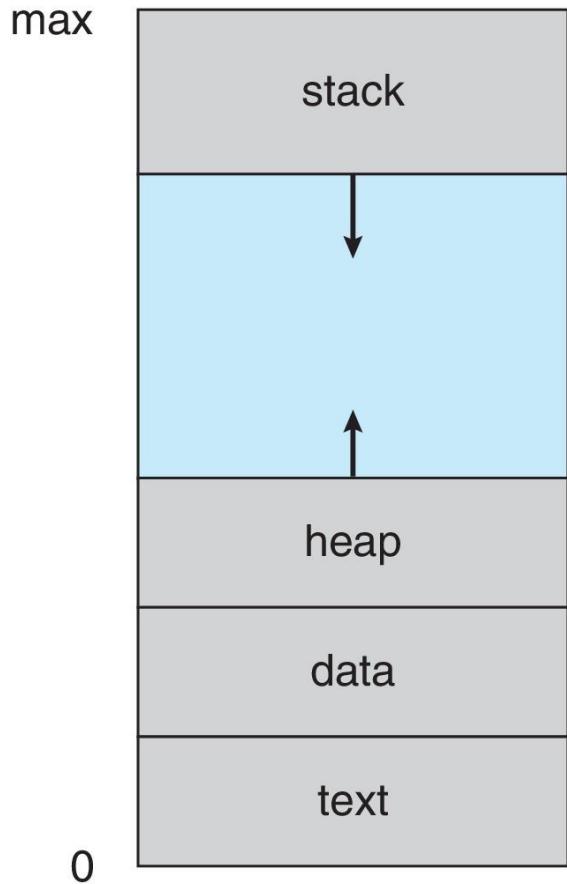
프로세스?

- 프로그램은 디스크(실행 파일)에 저장된 수동적 엔터티. 프로세스가 활성 상태.
 - 실행 파일이 메모리에 로드되면 프로그램이 프로세스가 된다.
- GUI 마우스 클릭, 명령줄 이름 입력 등을 통해 시작된 프로그램 실행
- 하나의 프로그램은 여러 프로세스가 될 수 있다.
 - 동일한 프로그램을 실행하는 여러 사용자를 고려.

1. Process Concept

Process in Memory

● Process in Memory

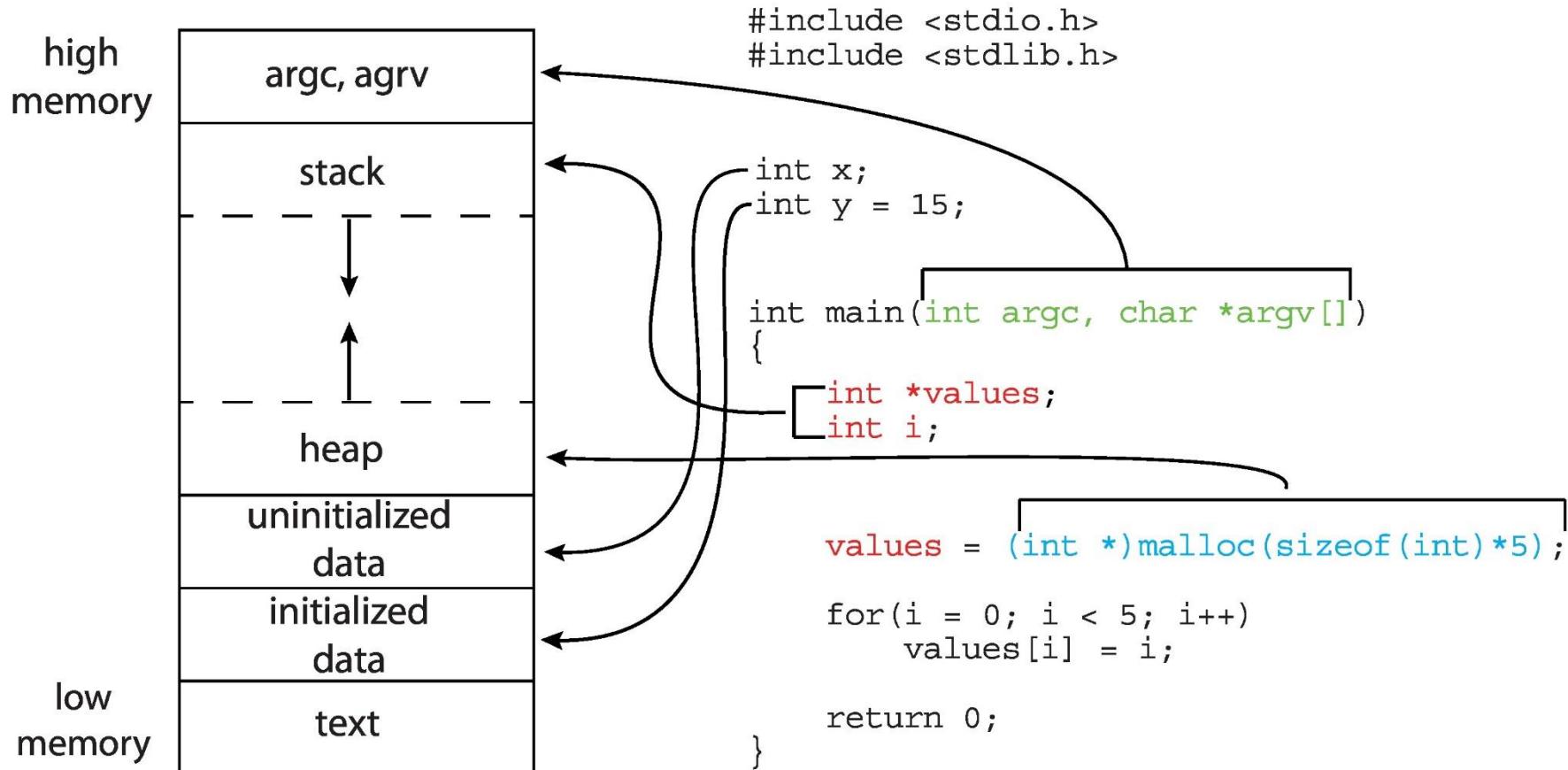


- **Text (Code) section:** 실행 가능한 코드
- **Data section:** 전역 변수
- **Heap section:** 프로그램이 실행되고 있는 런타임에 동적으로 할당되는 메모리 ex)
malloc, new
- **Stack section:** 함수 호출에 사용되는 임시적인 데이터를 저장함. (함수 매개변수, 리턴 주소, 지역 변수 등)

1. Process Concept

Process in Memory

- Memory Layout of a C Program





1. Process Concept

Process in Memory

● Memory Layout of a C Program

파일

소스코드

실습환경

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ cat > memory_layout.c
#include <stdio.h>
#include <stdlib.h>
```

```
int x;
int y = 15;
```

```
int main(int argc, char *argv[])
{
```

```
    int *values;
    int i;
```

```
    values = (int *)malloc(sizeof(int)*5);
```

```
    for(i=0; i<5; i++){
        values[i]=i;
    }
```

```
    return 0;
}
```

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ gcc memory_layout.c
```

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ ./a.out
```

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ size ./a.out
```

text	data	bss	dec	hex	filename
1418	604	12	2034	7f2	./a.out

결과값1

비고

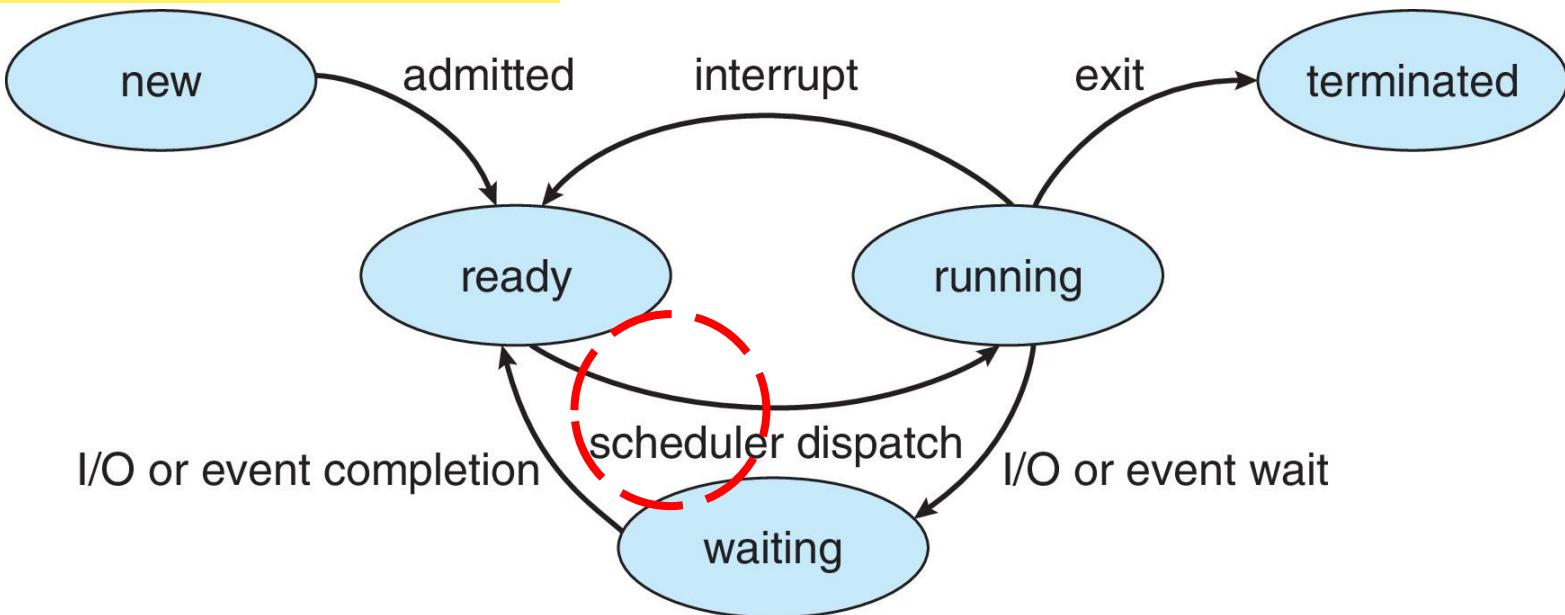
메모리 동적 할당과 배열을 활용. 위의 소스코드를 컴파일 하면 a.out이라는 실행 파일이 만들어지며 이를 프로그램이라 부른다. 그리고 프로그램이 위와 같은 메모리 레이아웃에 올려져 실행 중일 때, 그것을 프로세스라고 부른다. 이처럼 프로세스는 실행 중인 프로그램이다! 그리고 OS는 이러한 프로세스들을 관리한다.

1. Process Concept

Process State

- 프로세스가 실행되면 상태state가 변경.

- 1) 신규 New : 프로세스가 생성되고 있다.
 - 2) 실행 중 Running : 명령이 실행되고 있다.
 - 3) 대기 중 Waiting : 프로세스가 어떤 이벤트가 발생하기를 기다리고 있다.
 - 4) 준비 Ready : 프로세스가 프로세서에 할당되기를 기다리고 있다.
 - 5) 종료됨 Terminated : 프로세스가 실행을 완료.
- 프로세스 상태 다이어그램





1. Process Concept

Process Control Block (PCB)

- 각 프로세스와 관련된 정보(작업 제어 블록이라고도 함) 혹은 TCB(Task Control Block)이라고도 함

- 프로세스 상태 Process state – 실행 중, 대기 중 등
- 프로그램 카운터 Program counter – 다음에 실행할 명령의 위치
- CPU 레지스터 CPU registers – 모든 프로세스 중심 레지스터의 내용
- CPU 스케줄링 정보 CPU scheduling information - 우선 순위, 스케줄링 큐 포인터
- 메모리 관리 정보 Memory-management information – 프로세스에 할당된 메모리
- Accounting information – 사용된 CPU, 시작 이후 경과된 시계 시간, 시간 제한
- I/O 상태 정보 I/O status information – 프로세스에 할당된 I/O 장치, 열린 파일 목록

process state
process number
program counter
registers
memory limits
list of open files
...



1. Process Concept

Threads

- 지금까지 프로세스에는 단일 실행 스레드가 있다.

A process is a program that performs a single thread of execution.

- **프로세스당 여러 프로그램 카운터를 갖는 것을 고려.**

- 여러 위치에서 한 번에 실행할 수 있다.

- 제어의 다중 스레드 -> 스레드

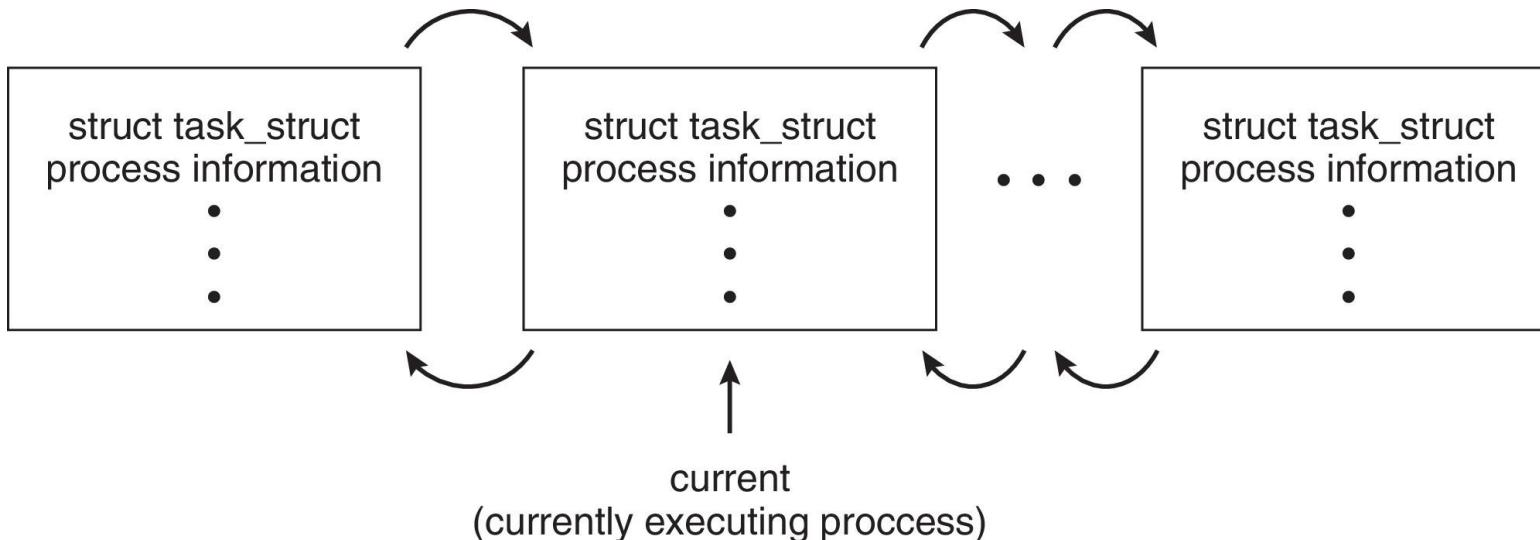
- 그런 다음 스레드 세부 정보, PCB의 여러 프로그램 카운터에 대한 스토리지가 있어야 한다.

1. Process Concept

Linux의 프로세스 표현

Represented by the C structure `task_struct`

```
pid t_pid;          /* process identifier */  
long state;         /* state of the process */  
unsigned int time_slice; /* scheduling information */  
struct task_struct *parent; /* this process's parent */  
struct list_head children; /* this process's children */  
struct files_struct *files; /* list of open files */  
struct mm_struct *mm;      /* address space of this process */
```





2. Process Scheduling

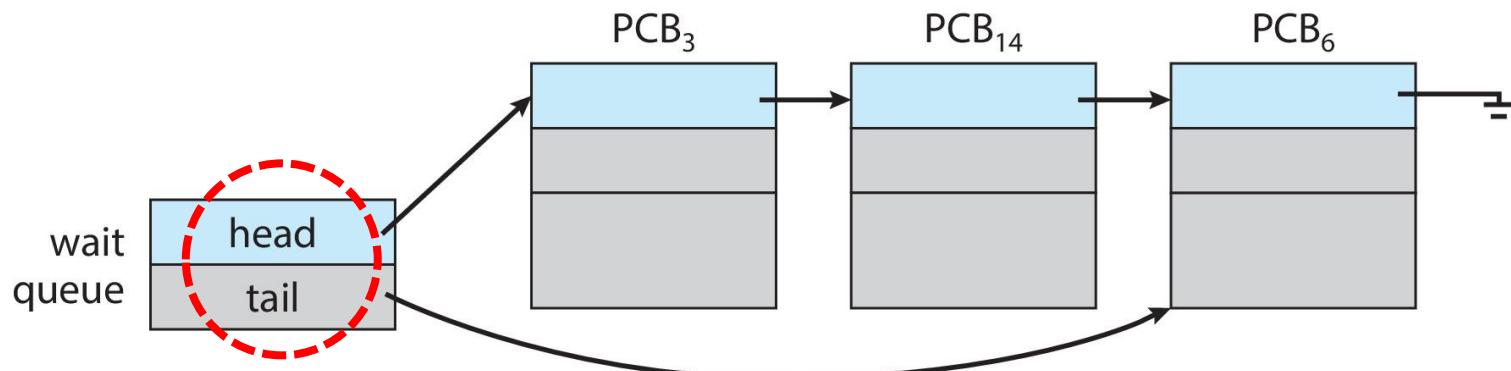
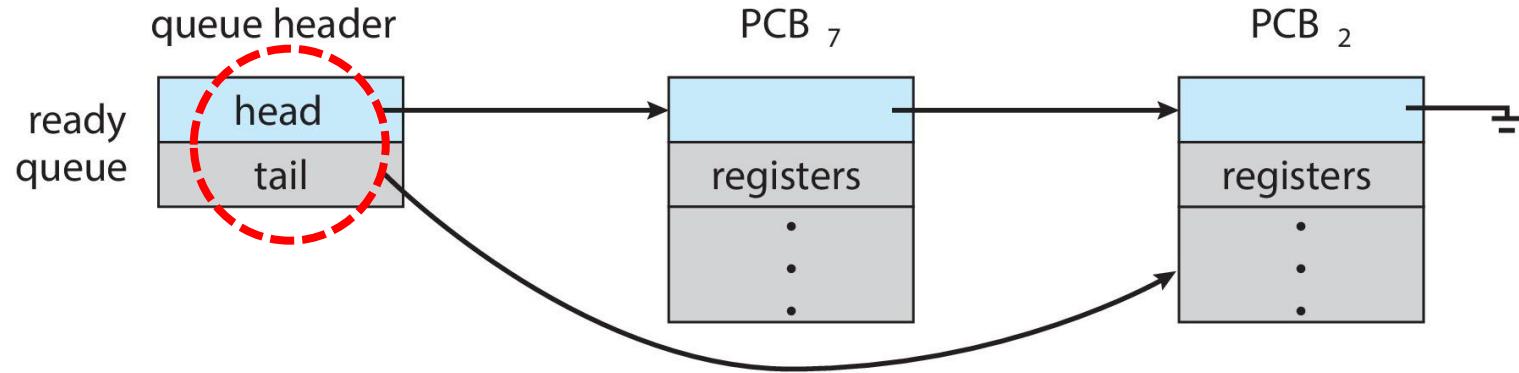
Process Scheduling?

- 프로세스 스케줄러는 CPU 코어에서 다음 실행을 위해 사용 가능한 프로세스 중에서 선택.
- 목표 -- CPU 사용 극대화, 프로세스를 CPU 코어로 신속하게 전환
- 프로세스의 스케줄링 대기열 **scheduling queues** 유지
 - 준비 큐 **Ready queue** – 메인 메모리에 상주하는 모든 프로세스의 집합으로, 실행 준비 및 대기
 - 대기 대기열 **Wait queues** - 이벤트(예: I/O)를 기다리는 프로세스 집합
 - 프로세스는 다양한 대기열 간에 마이그레이션.

2. Process Scheduling

Process Scheduling?

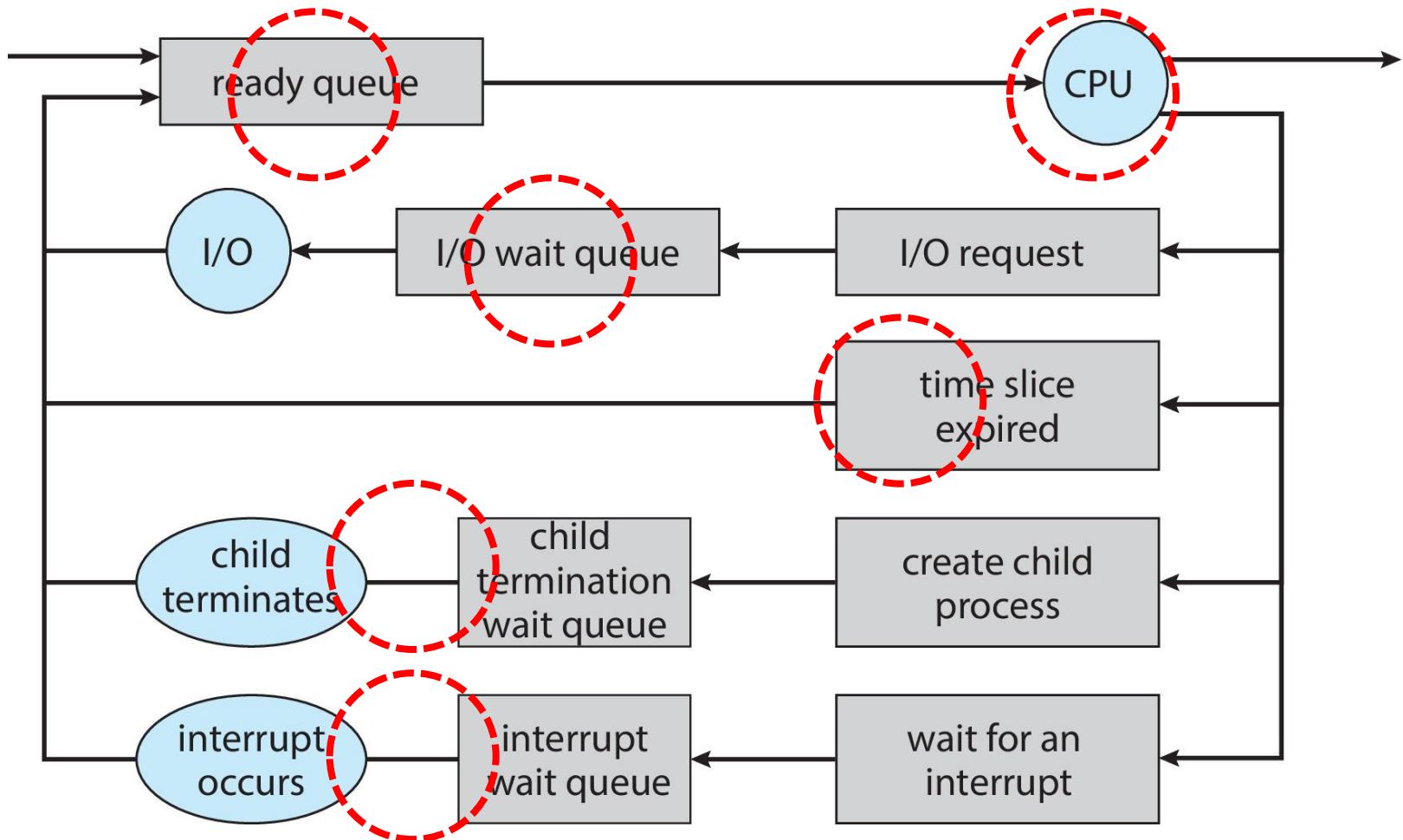
- Ready and Wait Queues 링크드 리스트로 구현 시



2. Process Scheduling

Process Scheduling?

- 프로세스 스케줄링의 표현





2. Process Scheduling

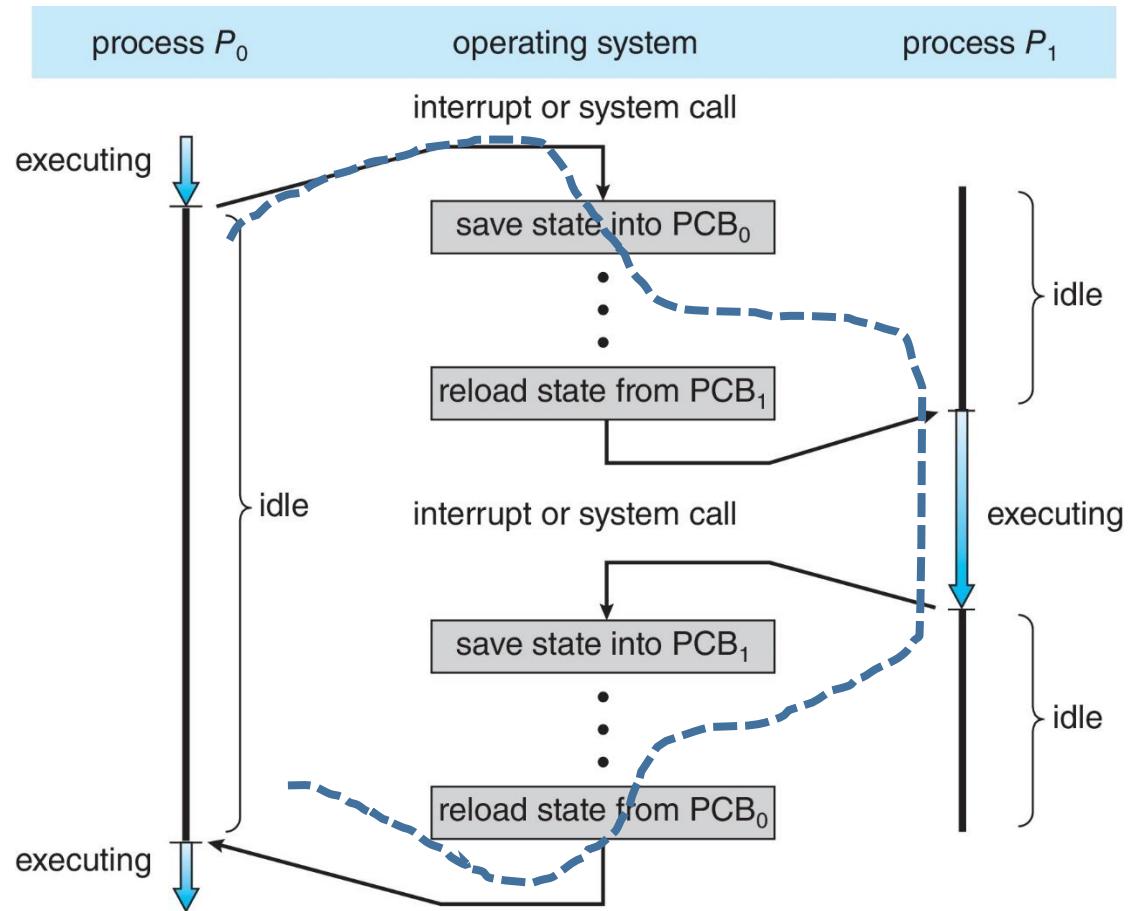
Context Switch

- CPU가 다른 프로세스로 전환할 때 시스템은 이전 프로세스의 상태를 저장하고 컨텍스트 전환을 통해 새 프로세스에 대해 저장된 상태를 로드해야 함.
- PCB에 표현된 프로세스의 컨텍스트
- 컨텍스트 전환 시간은 순수한 오버헤드. 전환하는 동안 시스템이 유용한 작업을 수행하지 않는다.
 - OS와 PCB가 복잡할수록 > 컨텍스트 스위치가 길어짐
 - 하드웨어 지원에 따라 시간이 달라짐
 - 일부 하드웨어는 CPU당 여러 레지스터 세트 제공 > 한 번에 여러 컨텍스트 로드

2. Process Scheduling

Process Scheduling?

- 프로세스에서 프로세스로 CPU 전환
 - 컨텍스트 전환 **context switch** 은 CPU가 한 프로세스에서 다른 프로세스로 전환할 때 발생.





2. Process Scheduling

모바일 시스템의 멀티태스킹

- 일부 모바일 시스템(예: 초기 iOS 버전)은 하나의 프로세스만 실행하도록 허용하고 다른 시스템은 일시 중지됨
- 화면 공간으로 인해 iOS가 제공하는 사용자 인터페이스 제한
 - 사용자 인터페이스를 통해 제어되는 단일 포그라운드 프로세스
 - 여러 백그라운드 프로세스 - 메모리에 있고 실행 중이지만 디스플레이에는 없고 제한이 있음
 - 제한에는 단일, 짧은 작업, 이벤트 알림 수신, 오디오 재생과 같은 특정 장기 실행 작업이 포함.
- **Android**는 더 적은 제한으로 포그라운드 및 백그라운드에서 실행.
 - 백그라운드 프로세스는 서비스를 사용하여 작업을 수행.
 - 백그라운드 프로세스가 일시 중단된 경우에도 서비스를 계속 실행할 수 있다.
 - 서비스에는 사용자 인터페이스가 없으며 메모리 사용량이 적다.



3. Operations on Processes

시스템 메커니즘 이슈

- 시스템은 다음에 대한 메커니즘을 제공.
 - 프로세스 생성
 - 프로세스 종료



3. Operations on Processes

Process Creation

- 부모 프로세스는 자식 프로세스를 만들고 자식 프로세스는 차례로 다른 프로세스를 만들어 프로세스 트리를 형성.
- 일반적으로 프로세스 식별자 **process identifier(pid)**를 통해 프로세스를 식별하고 관리.
- 리소스 공유 옵션
 - 부모와 자식이 모든 자원을 공유
 - 자녀는 부모 자원의 하위 집합을 공유.
 - 부모와 자식이 자원을 공유하지 않음
- 실행 옵션
 - 부모와 자식이 동시에 실행
 - 부모는 자식이 종료될 때까지 기다린다.

3. Operations on Processes

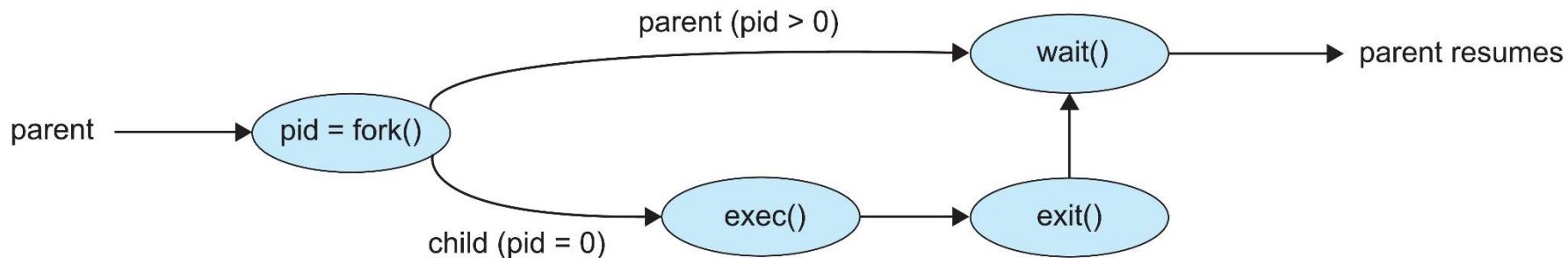
Process Creation

- 주소 공간

- 부모의 자식 복제
- 자식에 프로그램이 로드되어 있다.

- 유닉스 예제

- **fork()** 시스템 호출은 새로운 프로세스를 생성.
- 프로세스의 메모리 공간을 새 프로그램으로 대체하기 위해 **fork()** 다음에 사용되는 **exec()** 시스템 호출
- **자식 프로세스가 종료되기를 기다리는 부모 프로세스 호출 **wait()****

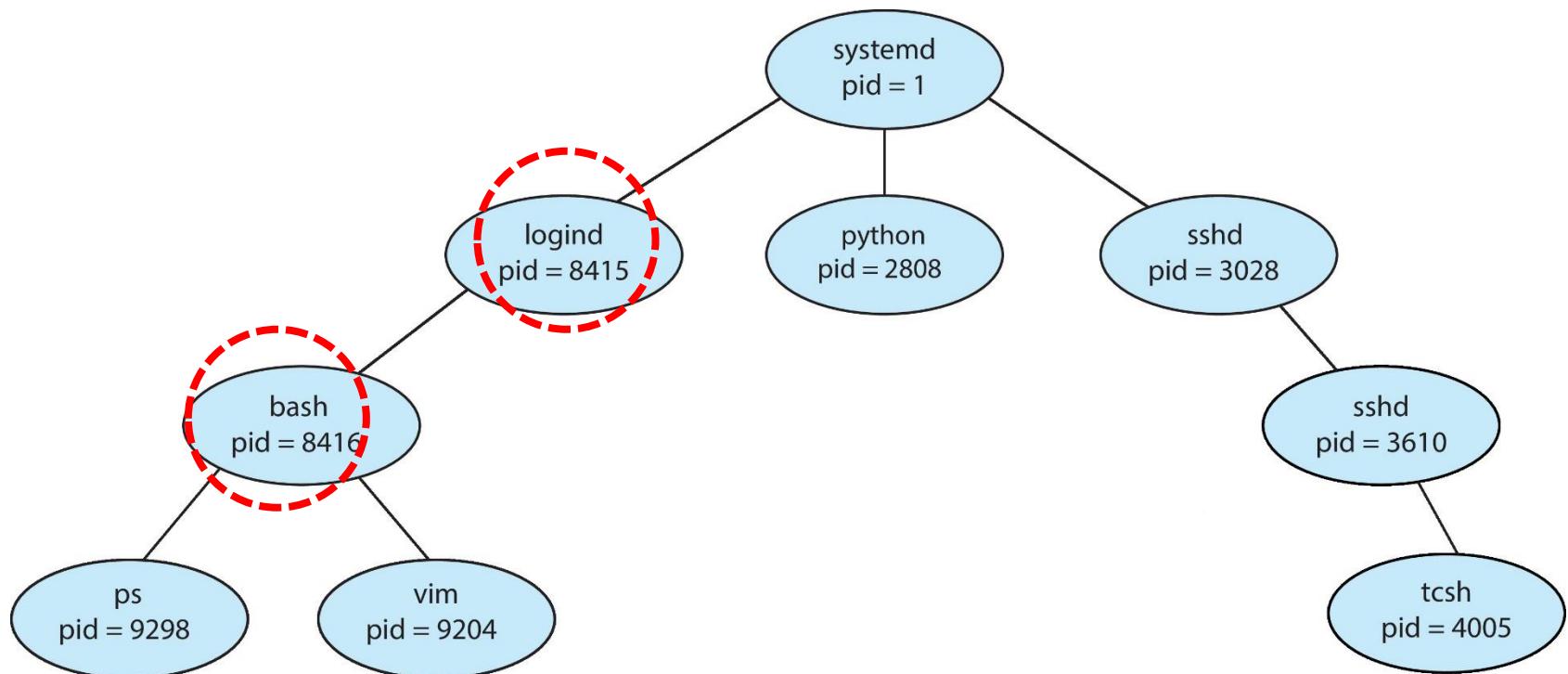




3. Operations on Processes

Process Creation

- Linux의 프로세스 트리





3. Operations on Processes

Process Creation

- C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execl("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

3. Operations on Processes

Process Creation

● C Program Forking Separate Process

파일

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ cat > fork_ex1.c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int main(){
    pid_t pid;

    // fork a child process
    pid = fork();

    if(pid < 0){ // error
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if(pid == 0){ // child process
        execlp("/bin/ls", "ls", NULL);
    }
    else{ // parent process
        // parent will wait for the child to complete.
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```

소스코드

결과값1

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ gcc fork_ex1.c
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ ./a.out
a.out fork_ex1.c memory_layout.c omp_1.c omp_2.c process_001.c
Child Completek8s@DESKTOP-RoEQ2U6:~/c_workspaces$
```

비고

- **fork()** 함수가 호출되어 자식 프로세스를 생성합니다. **fork()** 함수는 부모 프로세스와 자식 프로세스에서 서로 다른 값을 반환합니다. pid가 음수이면 오류가 발생한 것을 의미합니다. pid가 0이면 코드가 자식 프로세스에서 실행 중인 것입니다. pid가 양수이면 부모 프로세스에서 자식 프로세스의 프로세스 ID를 나타냅니다.
- if 문 내부에서 pid가 0보다 작은지 확인하여 fork() 함수에서 오류가 발생했는지를 검사합니다. 오류가 발생한 경우 fprintf()를 사용하여 오류 메시지를 stderr에 출력하고, 오류를 나타내는 1을 반환합니다.
- pid가 0인 경우 코드는 자식 프로세스에서 실행 중입니다. else if 블록 내에서 execlp() 함수가 호출되어 자식 프로세스를 ls 명령어로 대체합니다. execlp() 함수는 PATH 환경 변수에 지정된 디렉토리에서 명령어를 검색합니다. 이 경우에는 /bin/ls 명령어를 실행하며, 인자로 "ls"를 전달합니다.
- pid가 양수인 경우 코드는 부모 프로세스에서 실행 중입니다. else 블록 내에서 wait() 함수가 호출되어 부모 프로세스가 자식 프로세스의 완료를 기다립니다. NULL 인자는 자식 프로세스에 대한 상태 정보가 필요하지 않음을 의미합니다.
- 자식 프로세스가 완료되면 부모 프로세스는 실행을 계속하고, 콘솔에 "Child Complete"를 출력합니다.



3. Operations on Processes

Process Creation

● C Program Forking Separate Process

파일

소스코드

실행환경

소스코드

결과값1

- Two possibilities for execution
 - 부모 프로세스는 자식 프로세스와 함께 동시에 실행되거나, 자식 프로세스가 종료될 때까지 기다린다.
 - Two possibilities for address-space
 - 자식 프로세스는 부모 프로세스의 복사본이거나, 아예 새로운 프로그램을 로딩하는 경우도 있다.
- 비고
- `fork()`, `execvp()`, 그리고 `wait()` 함수를 사용하여 자식 프로세스를 생성하고, `execvp()`를 사용하여 `ls` 명령어를 실행한 다음 부모 프로세스가 자식 프로세스의 완료를 기다린 후에 "Child Complete"를 콘솔에 출력하는 예제



3. Operations on Processes

Process Creation

- Creating a Separate Process via Windows API

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

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
                      "C:\\\\WINDOWS\\\\system32\\\\mspaint.exe", /* command */
                      NULL, /* don't inherit process handle */
                      NULL, /* don't inherit thread handle */
                      FALSE, /* disable handle inheritance */
                      0, /* no creation flags */
                      NULL, /* use parent's environment block */
                      NULL, /* use parent's existing directory */
                      &si,
                      &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```



3. Operations on Processes

프로세스 종료 Termination

- Process는 마지막 명령문을 실행한 다음 exit() 시스템 호출을 사용하여 운영 체제에 삭제하도록 요청.
 - 자식에서 부모로 상태 데이터를 반환합니다(wait()를 통해).
 - 프로세스의 리소스는 운영 체제에 의해 할당 해제.
- 부모는 abort() 시스템 호출을 사용하여 자식 프로세스의 실행을 종료할 수 있다. 이렇게 하는 몇 가지 이유는 다음과 같다.
 - 자식이 할당된 리소스를 초과했다.
 - 자녀에게 할당된 작업이 더 이상 필요하지 않다.
 - 부모가 종료되고 운영 체제는 부모가 종료되면 자식이 계속되는 것을 허용하지 않다.



3. Operations on Processes

프로세스 종료 Termination

- 일부 운영 체제에서는 상위 항목이 종료된 경우 하위 항목의 존재를 허용하지 않는다. 프로세스가 종료되면 모든 하위 프로세스도 종료되어야 한다.
 - 계단식 종료. 모든 자녀, 손주 등이 종료.
 - 종료는 운영 체제에 의해 시작.
- 부모 프로세스는 `wait()` 시스템 호출을 사용하여 자식 프로세스의 종료를 기다릴 수 있다. 호출은 상태 정보와 종료된 프로세스의 pid를 반환.

PID = 대기(&상태);
- 기다리는 부모가 없으면(`wait()`를 호출하지 않은 경우) 프로세스는 좀비 **zombie** 이다.
- `wait()`를 호출하지 않고 부모가 종료된 경우 프로세스는 고아 **orphan** 이다.



3. Operations on Processes

fork() 예제

- 유닉스 라이크 운영체제에서 새로운 프로세스를 생성하는 시스템 콜
- fork()로 생성된 자식 프로세스는 부모 프로세스의 주소 공간을 복 사해옴
 - fork() 이후 두 프로세스는 각각 fork 이후의 명령어를 수행
 - fork()의 return 값이 0이면 자식 프로세스
 - return값이 0이 아닌 운영체제가 부여한 pid면 부모 프로세스, 즉 fork는 자식 프로세스의 PID를 반환함.
- wait()를 사용하면 부모 프로세스는 자식이 종료될 때까지 기다림



3. Operations on Processes

코드 - 주소 공간 복사

파일

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/바탕화면$ cd ~
```

```
k8s@DESKTOP-RoEQ2U6:~$ ls
```

공개 다운로드 문서 바탕화면 비디오 사진 음악 템플릿

```
k8s@DESKTOP-RoEQ2U6:~$ mkdir c-workspaces
```

실습환경

```
k8s@DESKTOP-RoEQ2U6:~$ ls
```

c-workspaces 공개 다운로드 문서 바탕화면 비디오 사진 음악 템플릿

```
k8s@DESKTOP-RoEQ2U6:~$ cd c-workspaces/
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ls
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > process_001.c
```

소스코드

결과값1

비고

3. Operations on Processes

코드 - 주소 공간 복사

파일

소스코드

실습환경

cat > process_001.c

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

```
int main()
```

```
{  
    pid_t pid;  
    pid = fork();  
  
    printf("Hello, Process!\n");  
  
    return 0;  
}
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ gcc process_001.c
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ls
```

```
a.out process_001.c
```

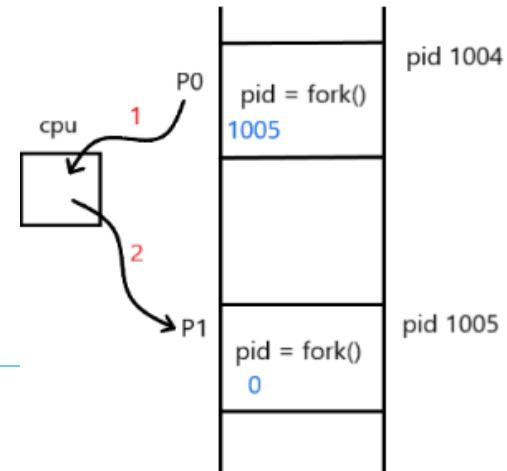
```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ./a.out
```

```
Hello, Process!
```

```
Hello, Process!
```

비고

- 자식 프로세스가 부모 프로세스의 주소 공간을 복사해오고 fork() 이후의 코드를 두 프로세스 다 실행하므로 Hello, Process가 두 번 출력됨





3. Operations on Processes

코드 - 리턴 값으로 구분

파일

소스코드

실습환경

```
k8s@8s@master1:~/c-workspaces$ cat > process_002.c
```

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

```
int main()
```

```
{
```

```
    pid_t pid;
    pid = fork();
```

```
    printf("Hello, Process! %d\n", pid);
```

```
    return 0;
```

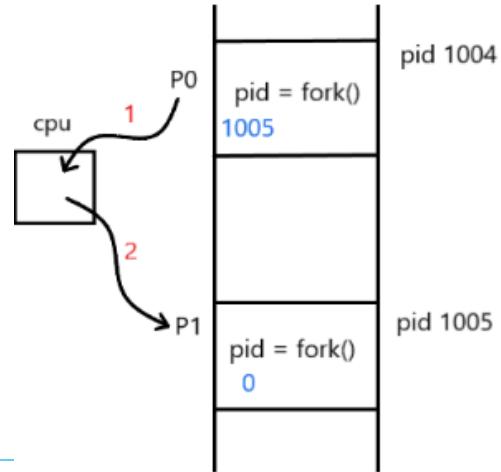
```
}
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ gcc process_002.c
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ./a.out
```

Hello, Process! 12359

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ Hello, Process! 0
```



결과값1

비고

- `fork()`에서 리턴 받음 `pid`를 출력하는데 부모는 os로부터 받은 자식의 `pid`를 출력하고 자식은 `0`을 출력함
- `fork()` 함수를 호출하여 자식 프로세스를 생성합니다. 부모 프로세스와 자식 프로세스는 이후의 코드를 동시에 실행하기 시작합니다. `fork()` 함수는 부모 프로세스에서는 자식 프로세스의 ID를, 자식 프로세스에서는 `0`을 반환합니다.
- "Hello, Process! %d\n" 형식 문자열을 출력합니다. `%d`는 `pid` 변수를 대체하는데, 부모 프로세스에서는 자식 프로세스의 ID가, 자식 프로세스에서는 `0`이 출력

3. Operations on Processes

코드 - wait()

파일

소스코드

실행환경

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > process_oo3.c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int main()
{
    pid_t pid;

    pid = fork();

    if (pid > 0){
        wait(NULL);
    }

    printf("Hello, Process! %d\n", pid);

    return 0;
}
```

wait(NULL) 함수
는 자식 프로세스가 종료될 때
까지 부모 프로세스를 대기

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ gcc process_oo3.c
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ./a.out
```

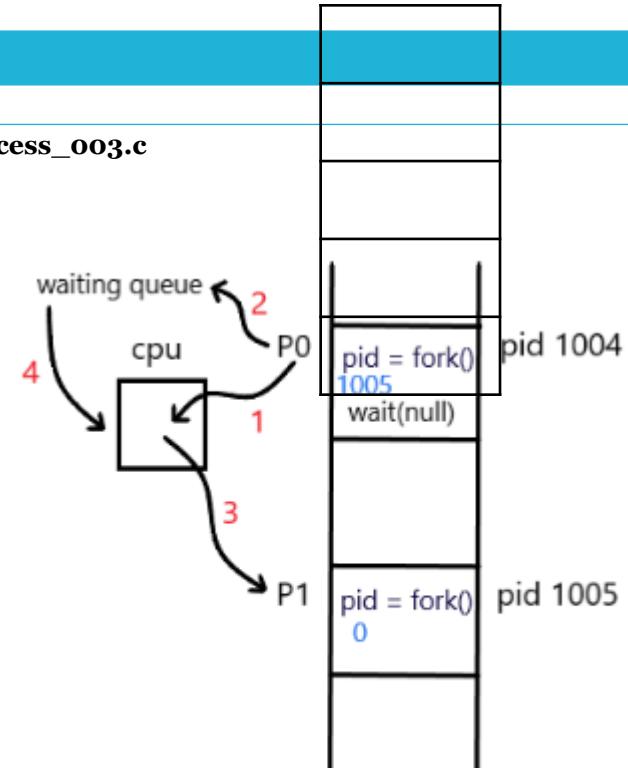
결과값1

Hello, Process! 0

Hello, Process! 12967

- 조건문을 통해 pid가 0이 아닐 부모 프로세스에 wait()를 걸음.
- 위에서 fork()만 하던 코드들과는 다르게 자식 프로세스가 종료된 후 부모 프로세스가 작업을 재개하므로 출력 결과를 보면 pid가 0인 자식이 먼저 printf했음을 알 수 있음

비고





3. Operations on Processes

출력이 무엇인지 설명하시오.

파일

소스코드

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > process_004.c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int value = 5;

int main()
{
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        value += 15;
        return 0;
    }
    else if (pid > 0) {
        wait(NULL);
        printf("Parent: value = %d\n", value);
    }
}
```

if 문을 사용하여 pid가 0인 경우에는 자식 프로세스에서 실행되는 부분입니다. 자식 프로세스에서는 value 변수에 15를 더 합니다.

else if 문을 사용하여 pid가 양수인 경우에는 부모 프로세스에서 실행되는 부분입니다. 부모 프로세스는 wait(NULL) 함수를 호출하여 자식 프로세스의 종료를 기다립니다. 그리고 "Parent: value = %d\n" 형식 문자열을 출력하여 value의 값을 출력합니다.

P0	5
P1	5

결과값1

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ gcc process_004.c
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ./a.out
```

```
Parent: value = 5
```

- 부모가 wait에 들어가고 자식이 먼저 15를 더해준 후 종료되고 부모가 출력을 수행하므로 20이라고 착각할 수 있음.
- 그러나 자식 프로세스가 주소 공간을 복사할 때 전역 변수 value 역시 복사되므로 자식의 value값만 변경되고 부모의 value는 변경되지 않는다.

비고



3. Operations on Processes

초기 부모 프로세스를 포함하여 생성된 프로세스 수

파일

소스코드

실행환경

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > process_005.c
```

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

int main()
{
    fork();
    fork();
    fork();

    printf("Hello, fork()!\n");
    return 0;
}
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ gcc process_005.c
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ./a.out
```

Hello, fork()!

Hello, fork()!

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ Hello, fork()
```

Hello, fork()!

Hello, fork()!

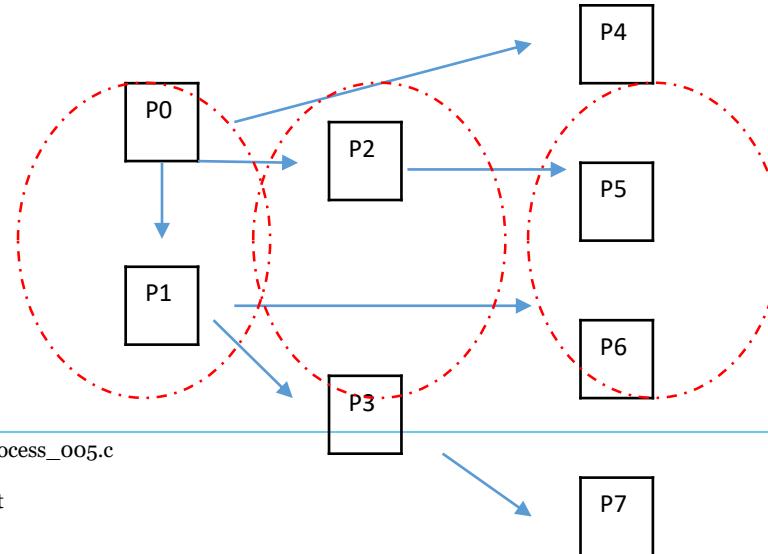
Hello, fork()!

Hello, fork()!

Hello, fork()!

결과값1

비고





3. Operations on Processes

초기 부모 프로세스를 포함하여 생성된 프로세스 수

파일

소스코드

실행환경

소스코드

결과값1

- 위의 코드를 실행했을 때 프로세스가 총 몇 개 생성되는지에 대한 문제이다.
- 부모는 총 3개의 자식을 생성한다.
- 첫 번째 fork로 생성된 자식은 2개
- 두 번째 fork로 생성된 자식은 1개
- 세 번째 fork로 생성된 자식은 남은 fork가 없으므로 0개
- 첫 번째 fork로 생성된 자식의 첫 fork로 생성된 자식은 1개의 자식을 생성할 수 있다.
- 첫 번째 fork로 생성된 자식의 두 번째 fork로 생성된 자식은 0개
- 두 번째 fork로 생성된 자식의 자식은 0개
- 다 더해주면 $3 + 2 + 1 + 1 = 7$
- 총 자식은 7개가 생성된다.
- 부모까지 총 8개의 프로세스가 생성된다.

비고



3. Operations on Processes

초기 부모 프로세스를 포함하여 생성된 프로세스 수

파일

소스코드

실행환경

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > process_oo6.c
```

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

int main()
{
    int i;
    pid_t pid;

    for (i = 0; i < 4; i++) {
        pid = fork();
    }
    printf("Hello, fork!\n");

    return 0;
}
```

소스코드

```
fork();
fork();
fork();
fork();
```

와 같음

결과값1

비고

3. Operations on Processes

초기 부모 프로세스를 포함하여 생성된 프로세스 수

파일

소스코드

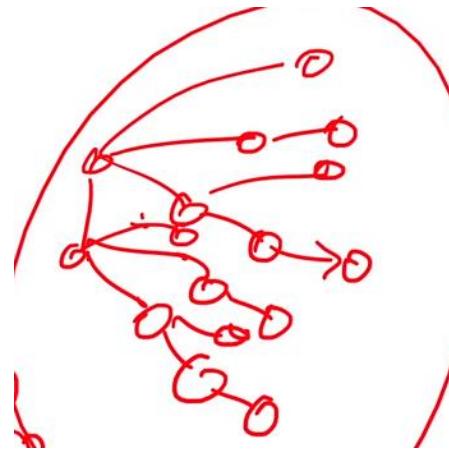
실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ gcc process_o06.c  
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ./a.out  
Hello, fork!  
Hello, fork!  
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ Hello, fork!  
Hello, fork!
```

결과값₁

• • •



- 총 16개가 만들어진다.
 - 자식 프로세스에서는 반복문이 처음부터 다시 시작되는 거 아닐까라고 생각할 수 있지만 지역 변수 `i`까지 같이 복사되므로 반복 횟수가 남아 있다.

비고



3. Operations on Processes

초기 부모 프로세스를 포함하여 생성된 프로세스 수

파일

소스코드

실습환경

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$  
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > process_007.c  
  
#include <stdio.h>  
#include <unistd.h>  
  
int main()  
{  
    int i;  
    pid_t pid;  
  
    for (i = 0; i < 4; i++) {  
        pid = fork();  
        printf("Hello, fork! %d\n", pid);  
    }  
  
    return 0;  
}
```

소스코드

결과값1

비고



3. Operations on Processes

초기 부모 프로세스를 포함하여 생성된 프로세스 수

파일

소스코드

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ gcc process_007.c
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ./a.out
```

```
Hello, fork! 34367
```

결과값1

```
Hello, fork! 34368
```

```
Hello, fork! 34369
```

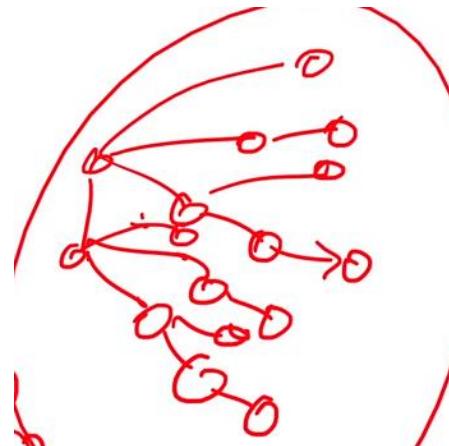
```
Hello, fork! 34370
```

```
Hello, fork! o
```

```
Hello, fork! 34371
```

...

비고





3. Operations on Processes

초기 부모 프로세스를 포함하여 생성된 프로세스 수

파일

소스코드

실습환경

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > process_oo8.c

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

int main()
{
    int value = 5;

    fork();
    value += 5;

    fork();
    value += 5;

    fork();
    value += 5;

    printf("Hello, fork! %d\n",value);

    return 0;
}
```

결과값1

비고



3. Operations on Processes

초기 부모 프로세스를 포함하여 생성된 프로세스 수

파일

소스코드

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ gcc process_oo8.c
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ./a.out
```

```
Hello, fork! 20
```

```
Hello, fork! 20
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ Hello, fork! 20
```

```
Hello, fork! 20
```

결과값1

비고

- 주어진 코드는 `fork()` 함수를 여러 번 호출하여 자식 프로세스를 생성하고, 각 자식 프로세스에서 `value` 변수에 5를 더한 후 최종 결과를 출력하는 예제입니다. 프로세스 생성 횟수에 따라 출력 결과가 중복되어 출력되게 됩니다. 출력 결과에는 `value` 변수의 최종 값인 20이 포함되어 있습니다.

3. Operations on Processes

초기 부모 프로세스를 포함하여 생성된 프로세스 수

파일

소스코드

실행환경

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > process_009.c

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

int main()
{
    int value = 5;

    fork();
    value += 5;
    printf("Hello, fork! %d\n",value);

    fork();
    value += 5;
    printf("Hello, fork! %d\n",value);

    fork();
    value += 5;
    printf("Hello, fork! %d\n",value);
    return 0;
}
```

소스코드

결과값1

비고

- 자식 프로세스의 수가 총 2개
- **value** 변수에 5를 더합니다. 이전에 부모 프로세스와 자식 프로세스에서 각각 **value**에 5를 더했으므로, 이제 **value**의 값은 10이 됩니다.

- 자식 프로세스의 수가 총 4개
- **value** 변수에 5를 더합니다. 이전에 부모 프로세스와 자식 프로세스에서 각각 **value**에 5를 더했으므로, 이제 **value**의 값은 15가 됩니다.

- 자식 프로세스의 수가 총 8개
- **value** 변수에 5를 더합니다. 이전에 부모 프로세스와 자식 프로세스에서 각각 **value**에 5를 더했으므로, 이제 **value**의 값은 20이 됩니다.



3. Operations on Processes

초기 부모 프로세스를 포함하여 생성된 프로세스 수

파일

소스코드

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ gcc process_009.c
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ./a.out
```

```
Hello, fork! 10
```

```
Hello, fork! 15
```

```
Hello, fork! 20
```

```
Hello, fork! 20
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ Hello, fork! 15
```

```
Hello, fork! 20
```

```
Hello, fork! 10
```

```
Hello, fork! 15
```

```
Hello, fork! 20
```

```
Hello, fork! 20
```

```
Hello, fork! 20
```

결과값1

비고



3. Operations on Processes

표시된 코드 줄이 발생한 상황을 설명하시오.

파일

소스코드

실습환경

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > process_010.c
```

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

int main()
{
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        execlp("/bin/ls", "ls", NULL);
        printf("LINE J\n");
    }

    else if (pid > 0) {
        wait(NULL);
        printf("Child Complete\n");
    }

    return 0;
}
```

소스코드

결과값1

비고

- if 문을 사용하여 pid가 0인 경우에는 자식 프로세스에서 실행되는 부분입니다. 자식 프로세스에서는 execlp() 함수를 호출하여 /bin/ls 명령을 실행합니다. execlp() 함수는 현재 프로세스를 새로운 프로세스로 대체합니다. 따라서 "LINE J\n" 문자열은 실행되지 않습니다.
- else if 문을 사용하여 pid가 양수인 경우에는 부모 프로세스에서 실행되는 부분입니다. 부모 프로세스는 wait(NULL) 함수를 호출하여 자식 프로세스의 종료를 기다립니다. 그리고 "Child Complete\n" 문자열을 출력합니다.

P0	
P1	ls



3. Operations on Processes

표시된 코드 줄이 발생한 상황을 설명하시오.

파일

소스코드

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ gcc process_o10.c
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ./a.out
```

결과값1

```
a.out      process_o02.c process_o04.c process_o06.c process_o08.c process_o10.c
```

```
process_o01.c process_o03.c process_o05.c process_o07.c process_o09.c
```

```
Child Complete
```

- **execlp는 fork로 프로세스의 주소 공간에 외부 프로그램을 덮어 씌움.**
- 이 경우 pid == 0인 자식 프로세스에 execlp 코드가 실행되므로 ls가 덮어씌워져 printf("LINE J\n"); 코드는 실행되지 않고 ls가 실행된다.

비고



3. Operations on Processes

프로그램을 사용하여 라인 A, B,C, D

파일

소스코드

실습환경

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > process_011.c
```

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

int main()
{
    pid_t pid, pid1;
    pid = fork();

    if (pid == 0) { // child process
        pid1 = getpid();
        printf("CHILD: pid = %d\n", pid); //A
        printf("CHILD: pid1 = %d\n", pid1); //B
    }

    else if (pid > 0) { // parent process
        wait(NULL);
        pid1 = getpid();
        printf("PARENT: pid = %d\n", pid); //C
        printf("PARENT: pid1 = %d\n", pid1); //D
    }

    return 0;
}
```

소스코드

결과값1

비고

B=C



3. Operations on Processes

프로그램을 사용하여 라인 A, B,C, D

파일

소스코드

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ gcc process_011.c  
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ ./a.out  
CHILD: pid = 0
```

결과값1

```
CHILD: pid1 = 723  
PARENT: pid = 723  
PARENT: pid1 = 722  
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$
```

- **getpid()**는 현재 실행 중인 프로세스의 pid를 얻는다.
- 부모 프로세스의 pid를 723 할 때
- A는 0 자식이 fork로부터 반환 받은 0
- B는 723 getpid()로부터 반환 받은 자식 프로세스 자신의 pid
- C는 723 fork로부터 반환 받은 자식 프로세스의 pid
- D는 722 getpid()로부터 반환 받은 부모 프로세스 자신의 pid이다.

비고



3. Operations on Processes

출력이 무엇인지 설명하시오.

파일

소스코드

실습환경

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > process_012.c
#include <wait.h>
#include <stdio.h>
#include <unistd.h>

#define SIZE 5
int nums[SIZE] = {0,1,2,3,4};

int main()
{
    int i;
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        for (i = 0; i < SIZE; i++) {
            nums[i] *= i;
            printf("CHILD: %d \n",nums[i]); /* LINE X */
        }
    }

    else if (pid > 0) {
        wait(NULL);
        for (i = 0; i < SIZE; i++) {
            printf("PARENT: %d \n",nums[i]); /* LINE Y */
        }
    }
    return 0;
}
```

결과값1

비고



3. Operations on Processes

출력이 무엇인지 설명하시오.

파일

소스코드

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ gcc process_012.c
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ./a.out
```

```
CHILD: 0
```

```
CHILD: 1
```

```
CHILD: 4
```

```
CHILD: 9
```

```
CHILD: 16
```

```
PARENT: 0
```

```
PARENT: 1
```

```
PARENT: 2
```

```
PARENT: 3
```

```
PARENT: 4
```

• 사실상 자식 프로세스는 i의 제곱 수, 부모는 i를 그대로 반환하게 된다.

• 자식 프로세스에서 수정된 `nums` 배열의 각 요소가 출력

• 부모 프로세스에서 자식 프로세스와 공유되는 `nums` 배열의 각 요소가 출력

결과값1

비고



3. Operations on Processes

Android 프로세스 중요도 계층

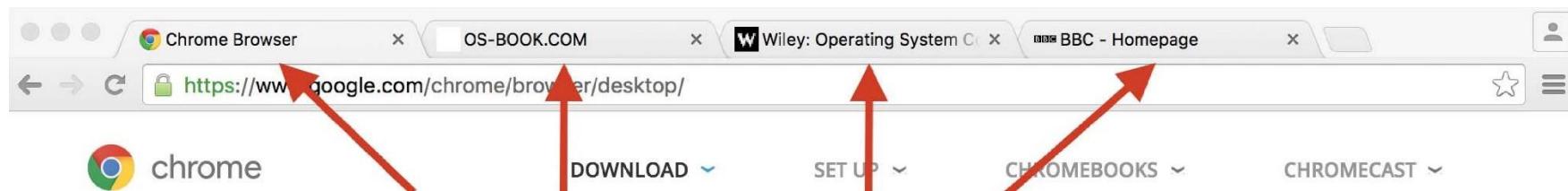
- 모바일 운영 체제는 종종 메모리와 같은 시스템 리소스를 회수하기 위해 프로세스를 종료. 가장 중요한 것부터 가장 중요하지 않은 것까지:
 - 전경 프로세스
 - 보이는 과정
 - 서비스 프로세스
 - 백그라운드 프로세스
 - 빈 프로세스
- Android는 가장 덜 중요한 프로세스를 종료하기 시작.



3. Operations on Processes

Multiprocess Architecture – Chrome Browser

- 많은 웹 브라우저가 단일 프로세스로 실행(일부는 여전히 실행됨).
 - 하나의 웹 사이트가 문제를 일으키면 전체 브라우저가 중단되거나 충돌할 수 있다.
- **Google Chrome** 브라우저는 3가지 유형의 프로세스가 있는 다중 프로세스.
 - 브라우저 프로세스는 사용자 인터페이스, 디스크 및 네트워크 I/O를 관리.
 - 렌더러 프로세스는 웹 페이지를 렌더링하고 HTML, Javascript를 처리 합니다. 열린 각 웹사이트에 대해 생성된 새 렌더러
 - ✓ 디스크 및 네트워크 I/O를 제한하는 샌드박스 **sandbox**에서 실행하여 보안 익스플로잇의 영향 최소화
 - 플러그인 유형별 플러그인 프로세스



Each tab represents a separate process.



4. Interprocess Communication

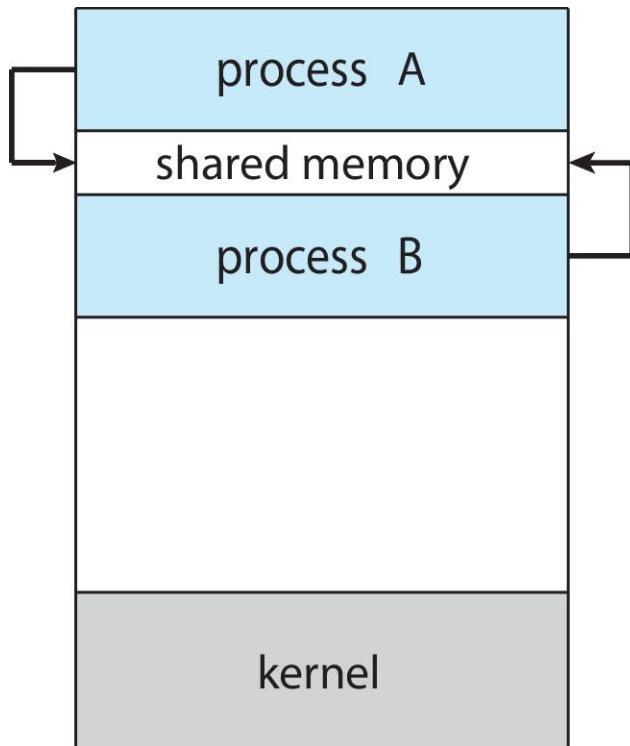
프로세스 간 통신

- 시스템 내의 프로세스는 독립적 independent 이거나 협력적 cooperative 일 수 있다.
- 협력 프로세스는 데이터 공유를 포함하여 다른 프로세스에 영향을 미치거나 영향을 받을 수 있다.
- 협력 프로세스의 이유:
 - 정보 공유
 - 계산 속도 향상
 - 모듈성
 - 편의
- 협력 프로세스에는 IPC(interprocess communication 프로세스 간 통신)가 필요.
- IPC의 두 가지 모델
 - 공유 메모리 Shared memory
 - 메시지 전달 Message passing

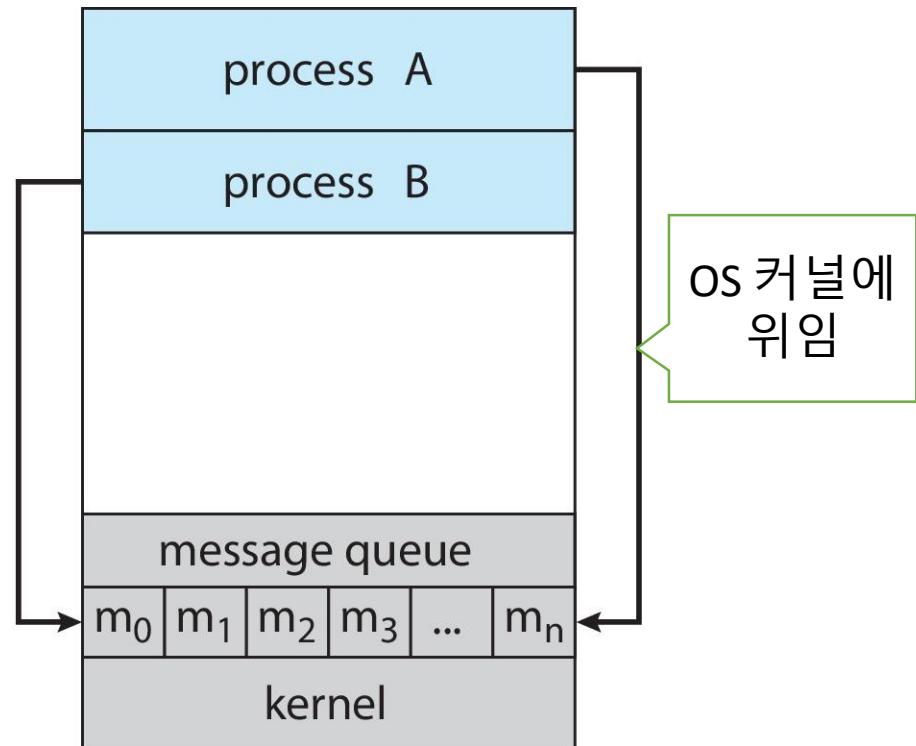
4. Interprocess Communication

Communications Models

(a) Shared memory.



(b) Message passing.



(a)

(b)



4. Interprocess Communication

생산자-소비자 Producer-Consumer 문제

- 협력 프로세스를 위한 패러다임:
 - 생산자 프로세스는 소비자 프로세스에서 소비되는 정보를 생성.
- 두 가지 변형:
 - 무제한 버퍼 **unbounded-buffer** 는 버퍼 크기에 실질적인 제한을 두지 않는다.
 - ① 생산자는 절대 기다리지 않는다
 - ② 소비할 버퍼가 없으면 소비자가 기다린다.
 - **bounded-buffer**는 고정된 버퍼 크기가 있다고 가정.
 - ① 모든 버퍼가 가득 차면 생산자는 기다려야 한다.
 - ② 소비할 버퍼가 없으면 소비자가 기다린다.



4. Interprocess Communication

IPC – Shared Memory

- 통신하려는 프로세스 간에 공유되는 메모리 영역
- 통신은 운영 체제가 아닌 사용자 프로세스의 제어를 받는다.
- 주요 문제는 사용자 프로세스가 공유 메모리에 액세스할 때 작업을 동기화할 수 있는 메커니즘을 제공하는 것.

제한된 버퍼 – 공유 메모리 솔루션

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
```

```
item buffer[BUFFER_SIZE];
```

```
int in = 0;
int out = 0;
```

- 솔루션은 정확하지만 BUFFER_SIZE-1 요소만 사용할 수 있다.



4. Interprocess Communication

생산자 프로세스 – 공유 메모리

```
item next_produced;  
  
while (true) {  
    /* produce an item in next produced */  
    while (((in + 1) % BUFFER_SIZE) == out)  
        ; /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
}
```

대기

$((in + 1) \% \text{BUFFER_SIZE}) == \text{out}$ 조건은
버퍼가 가득 차 있는지를 확인합니다.
즉, 버퍼의 다음 위치가 out 위치와 동
일하면 버퍼가 가득 찬 상태입니다.

소비자 프로세스 – 공유 메모리

```
item next_consumed;  
  
while (true) {  
    while (in == out)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) \% BUFFER_SIZE;  
  
    /* consume the item in next consumed */  
}
```

대기



4. Interprocess Communication

모든 버퍼를 채우는 것

- 모든 버퍼를 채우는 소비자-생산자 문제에 대한 솔루션을 제공하고 싶다고 가정.
- 전체 버퍼 수를 추적하는 정수 카운터 `counter`를 사용하여 그렇게 할 수 있다.
- 처음에는 카운터 `counter`가 0으로 설정.
- 정수 카운터 `counter`는 새 버퍼를 생성한 후 생산자에 의해 증가.
- 정수 카운터 `counter`는 버퍼를 소비한 후 소비자에 의해 감소.



4. Interprocess Communication

Producer

```
while (true) {  
    /* produce an item in next produced */  
  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

Consumer

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next consumed */  
}
```



4. Interprocess Communication

경쟁 조건Race Condition

counter++ could be implemented as

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

counter-- could be implemented as

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

"counter++" 연산은 먼저 수행되고 그 다음에 "counter--" 연산이 수행됩니다. 따라서 초기에 "count" 변수의 값이 5 이므로, "counter++" 연산 후에 "count" 변수의 값은 6이 되고, 그 다음에 "counter--" 연산이 수행되어 "count" 변수의 값은 4가 됩니다.

Consider this execution interleaving with “count = 5” initially:

So: producer execute **register1 = counter** {register1 = 5}

S1: producer execute **register1 = register1 + 1** {register1 = 6}

S2: consumer execute **register2 = counter** {register2 = 5}

S3: consumer execute **register2 = register2 - 1** {register2 = 4}

S4: producer execute **counter = register1** {counter = 6 }

S5: consumer execute **counter = register2** {counter = 4}



4. Interprocess Communication

IPC – Message Passing

- 프로세스는 공유 변수에 의존하지 않고 서로 통신.
- IPC 시설은 두 가지 작업을 제공.
 - **send(message)**
 - **receive(message)**
- 메시지 크기는 고정되거나 가변적.



4. Interprocess Communication

IPC – Message Passing

- 프로세스 P와 Q가 통신하려면 다음을 수행.
 - 그들 사이에 통신 링크를 설정
 - 보내기/받기를 통한 메시지 교환
- 구현 문제:
 - 링크는 어떻게 설정될까?
 - 링크를 세 개 이상의 프로세스와 연관시킬 수 있을까?
 - 모든 통신 프로세스 쌍 사이에 몇 개의 링크가 있을 수 있을까?
 - 링크의 용량은 얼마일까?
 - 링크가 수용할 수 있는 메시지의 크기는 고정입니까 아니면 가변인가?
 - 링크는 단방향일까, 양방향일까?



4. Interprocess Communication

통신 링크 Communication Link 구현

- 물리적 Physical :
 - 공유 메모리
 - 하드웨어 버스
 - 회로망 Network
- 논리Logical:
 - 직접 또는 간접
 - 동기식 또는 비동기식
 - 자동 또는 명시적 버퍼링



4. Interprocess Communication

Direct Communication

- 프로세스는 서로 명시적으로 이름을 지정.
 - send (P, message) – 프로세스 P에게 메시지를 보낸다.
 - receive(Q, message) – 프로세스 Q로부터 메시지 수신
- 통신 링크의 속성
 - 링크가 자동으로 설정.
 - 링크는 정확히 한 쌍의 통신 프로세스와 연결.
 - 각 쌍 사이에는 정확히 하나의 링크가 있다.
 - 링크는 단방향일 수 있지만 일반적으로 양방향.



4. Interprocess Communication

Indirect Communication

- 메시지는 **mailbox**(포트 ports 라고도 함)에서 전달 및 수신.
 - 각 mailbox에는 고유한 ID가 있다.
 - 프로세스는 mailbox을 공유하는 경우에만 통신할 수 있다.
- 통신 링크의 속성
 - 프로세스가 공통 mailbox을 공유하는 경우에만 링크 설정
 - 링크는 많은 프로세스와 연결될 수 있다.
 - 각 프로세스 쌍은 여러 통신 링크를 공유할 수 있다.
 - 링크는 단방향 또는 양방향일 수 있다.
- 운영
 - 새 mailbox(포트 ports) 만들기
 - mailbox을 통해 메시지 보내기 및 받기
 - mailbox 삭제
- 프리미티브는 다음과 같이 정의.
 - **send(A, message)** – mailbox A로 메시지 보내기
 - **receive(A, message)** – mailbox A에서 메시지 수신



4. Interprocess Communication

Indirect Communication

- **mailbox 공유**

- P₁, P₂ 및 P₃ 공유 mailbox A
- P₁, 보낸다; P₂ 및 P₃ 수신
- 누가 메시지를 받습니까?

- **솔루션**

- 링크가 최대 두 개의 프로세스와 연결되도록 허용
- 수신 작업을 실행하기 위해 한 번에 하나의 프로세스만 허용
- 시스템이 수신기를 임의로 선택하도록 허용합니다. 보낸 사람은 받는 사람이 누구인지 알린다.



4. Interprocess Communication

Synchronization

- 메시지 전달은 차단 또는 비차단일 수 있다.
- 차단 Blocking 은 동기식synchronous으로 간주.
 - 보내기 차단 Blocking send -- 메시지가 수신될 때까지 보낸 사람이 차단.
 - 수신 차단 Blocking receive -- 메시지를 사용할 수 있을 때까지 수신자가 차단.
- 비 차단 Non-blocking 은 비동기asynchronous로 간주.
 - Non-blocking send -- 보낸 사람이 메시지를 보내고 계속.
 - 비차단 수신 Non-blocking receive -- 수신자는 다음을 수신.
 - ✓ 유효한 메시지 또는 널 메시지
- 다양한 조합 가능
- 송신과 수신이 모두 차단되면 랑데부 rendezvous 가 있는 것.



4. Interprocess Communication

생산자-소비자: 메시지 전달

Producer

```
message next_produced;  
while (true) {  
    /* produce an item in next_produced */  
  
    send(next_produced);  
}
```

Consumer

```
message next_consumed;  
while (true) {  
    receive(next_consumed)  
  
    /* consume the item in next_consumed */  
}
```



4. Interprocess Communication

Buffering

- 링크에 첨부된 메시지 대기열 Queue.
- 세 가지 방법 중 하나로 구현
 - 1. 제로 용량 – 링크에 대기 중인 메시지가 없다.
 - 발신자는 수신자를 기다려야 함(랑데부 rendezvous)
 - 2. 제한된 용량 – n개 메시지의 유한 길이
 - 링크가 가득 차면 발신자는 기다려야 함
 - 3. 무한한 용량 – 무한한 길이
 - Sender는 절대 기다리지 않는다.



4. Interprocess Communication

Buffering

- 세 가지 방법 중 하나로 메시지 대기열 Queue를 구현할 수 있다.
 - 제로 용량 (Zero Capacity):
제로 용량 방식에서는 링크에 대기 중인 메시지가 없으며, 발신자는 수신자를 기다려야 합니다. 이는 랭데부 (rendezvous)라고도 알려져 있습니다. 발신자와 수신자가 동시에 동작하여 메시지를 교환하며, 발신자는 수신자가 준비될 때까지 대기합니다. 수신자는 메시지를 받고 처리한 후에 다음 메시지를 받을 준비를 합니다.
 - 제한된 용량 (Limited Capacity):
제한된 용량 방식에서는 링크에는 최대 n개의 메시지만 저장할 수 있습니다. 링크가 가득 차면 발신자는 기다려야 합니다. 이 방식은 일반적으로 고정된 크기의 버퍼나 배열을 사용하여 구현됩니다. 발신자는 메시지를 큐에 추가할 때 링크가 가득 차 있는지 확인하고, 가득 차 있을 경우 대기합니다. 수신자는 메시지를 받고 처리한 후, 큐에서 다음 메시지를 꺼내어 처리합니다.
 - 무한한 용량 (Infinite Capacity):
무한한 용량 방식에서는 링크에는 용량 제한이 없으며, 발신자는 절대로 기다리지 않습니다. 발신자는 메시지를 큐에 추가하고 즉시 다음 작업으로 넘어갑니다. 수신자는 메시지를 받고 처리하는데 필요한 시간이 걸리더라도, 발신자는 수신자의 상태에 대해 알지 못하고 계속 다음 메시지를 전송합니다. 이 방식은 일반적으로 큐의 크기를 동적으로 조정하여 구현됩니다.
- 세 가지 방법은 메시지 대기열을 다루는 상황에 따라 선택될 수 있으며, 용량 요구 사항과 동기화의 필요성에 따라 결정됩니다.



5. Examples of IPC Systems

IPC 시스템의 예 - POSIX

- **POSIX(Portable Operating System Interface(for UNIX) 공유 메모리**

- **프로세스는 먼저 공유 메모리 shared memory 세그먼트를 생성.**

`shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`

- 기존 세그먼트를 여는 데에도 사용됨
- 개체의 크기 설정

`ftruncate(shm_fd, 4096);`

- `mmap()`을 사용하여 파일 포인터를 공유 메모리 객체에 메모리 맵핑
- 공유 메모리에 대한 읽기 및 쓰기는 `mmap()`에 의해 반환된 포인터를 사용하여 수행.

5. Examples of IPC Systems

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* strings written to shared memory */
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory obect */
    void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

5. Examples of IPC Systems

IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory obect */
    void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s", (char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```



5. Examples of IPC Systems

POSIX Shared Memory

파일

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > shm_producer.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main()
{
    const int SIZE = 4096;           // shared memory의 크기
    const char* name = "OS";        // shared memory의 이름
    const char* message_o = "Hello, ";
    const char* message_i = "Shared Memory!\n";

    int shm_fd; // shared memory의 file 설명자
    char* ptr; // shared memory의 pointer

    // shared memory 객체 생성
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    // shared memory의 크기 설정
    ftruncate(shm_fd, SIZE);

    // shared memory 객체에 매핑
    ptr = (char*) mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);

    // shared memory에 메시지 작성
    sprintf(ptr, "%s", message_o);
    ptr += strlen(message_o);
    sprintf(ptr, "%s", message_i);
    ptr += strlen(message_i);

    return 0;
}
```

결과값1

비고



5. Examples of IPC Systems

POSIX Shared Memory

파일

소스코드

실습환경

소스코드

결과값1

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ gcc shm_producer.c -lrt -o shared_producer.out
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ./shared_producer.out
```

비고

- 위와 같이 실행하면 겉으로 보기에는 아무 변화도 없을 것입니다. 하지만 **shared_producer.out** 실행 프로그램을 실행한 순간 공유 메모리에 메시지를 저장하였습니다.



5. Examples of IPC Systems

POSIX Shared Memory

파일

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > shm_customer.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main()
{
    const int SIZE = 4096;           // shared memory의 크기
    const char* name = "OS";        // shared memory의 이름

    int shm_fd; // shared memory의 file 설명자
    char* ptr; // shared memory의 pointer

    // shared memory 객체 생성
    shm_fd = shm_open(name, O_RDONLY, 0666);

    // shared memory 객체에 매핑
    ptr = (char*) mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    // shared memory 객체 읽기
    printf("%s", (char*)ptr);

    // shared memory 제거
    shm_unlink(name);

    return 0;
}
```

결과값1

비고



5. Examples of IPC Systems

POSIX Shared Memory

파일

소스코드

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ gcc shm_customer.c -lrt -o shared_customer.out
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ./shared_customer.out
```

결과값1

```
Hello, Shared Memory!
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$
```

- 생산자가 메시지를 생산하였기 때문에 아래의 소비자 프로세스를 구현하여 메시지를 읽습니다.
- 위와 같이 `shared_customer.out` 실행 프로그램을 수행하면 메시지를 읽을 수 있습니다. 다시한번 프로그램을 수행시키면 더이상 읽을 수 있는 메시지가 없어서 `Segmentation fault` 메시지를 받습니다. 메시지를 다시 읽기 위해서는 생산자 프로그램이 수행되어야 합니다.

비고



5. Examples of IPC Systems

Examples of IPC Systems - Mach

- Mach 통신은 메시지 기반.
- 시스템 콜도 메시지다
- 각 태스크는 생성 시 두 개의 포트(Kernel 및 Notify)를 얻는다.
- 메시지는 `mach_msg()` 함수를 사용하여 보내고 받는다.
- 다음을 통해 생성된 통신에 필요한 포트

`mach_port_allocate()`

- 보내기 및 받기는 유연. 예를 들어 사서함이 가득 찬 경우 네 가지 옵션:
 - 무기한 대기
 - 최대 n밀리초 대기
 - 즉시 반환
 - 일시적으로 메시지 캐시



5. Examples of IPC Systems

Mach Messages

```
#include<mach/mach.h>

struct message {
    mach_msg_header_t header;
    int data;
};

mach port t client;
mach port t server;
```



5. Examples of IPC Systems

Mach Message Passing - Client

```
/* Client Code */

struct message message;

// construct the header
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;

// send the message
mach_msg(&message.header, // message header
         MACH_SEND_MSG, // sending a message
         sizeof(message), // size of message sent
         0, // maximum size of received message - unnecessary
         MACH_PORT_NULL, // name of receive port - unnecessary
         MACH_MSG_TIMEOUT_NONE, // no time outs
         MACH_PORT_NULL // no notify port
);
```



5. Examples of IPC Systems

Mach Message Passing - Server

```
/* Server Code */

struct message message;

// receive the message
mach_msg(&message.header, // message header
         MACH_RCV_MSG, // sending a message
         0, // size of message sent
         sizeof(message), // maximum size of received message
         server, // name of receive port
         MACH_MSG_TIMEOUT_NONE, // no time outs
         MACH_PORT_NULL // no notify port
);
```



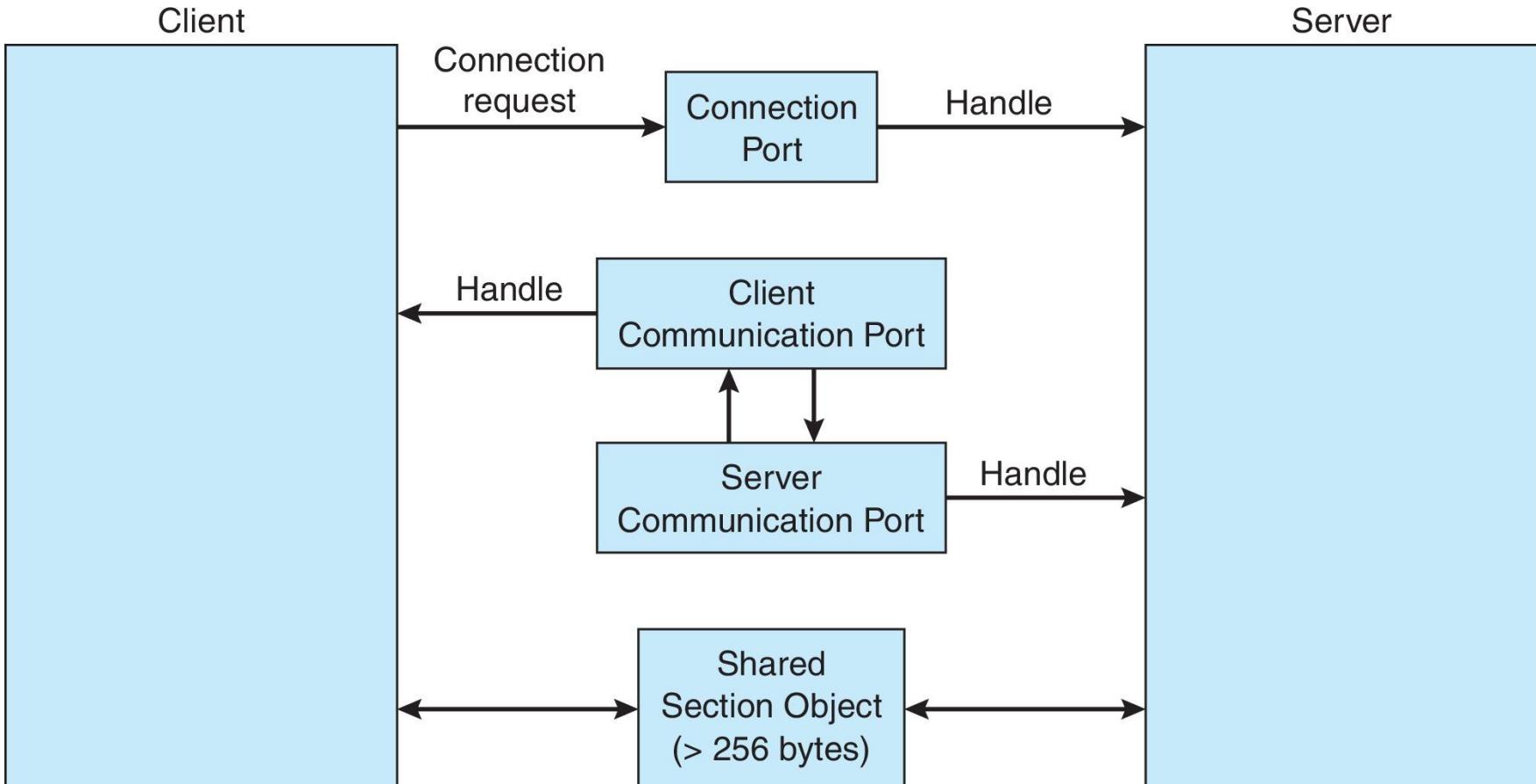
5. Examples of IPC Systems

Examples of IPC Systems – Windows

- 고급 로컬 절차 호출 **advanced local procedure call (LPC)** 기능을 통한 메시지 전달 중심
- 동일한 시스템의 프로세스 간에만 작동
- 포트 ports (사서함 mailboxes 과 같은)를 사용하여 통신 채널을 설정하고 유지.
- 통신은 다음과 같이 작동.
 - 클라이언트는 하위 시스템의 연결 포트 **connection port** 개체에 대한 핸들을 염.
 - 클라이언트가 연결 요청을 보냄.
 - 서버는 두 개의 개인 통신 포트 **communication ports** 를 만들고 그 중 하나에 대한 핸들을 클라이언트에 반환.
 - 클라이언트와 서버는 해당 포트 핸들을 사용하여 메시지 또는 콜백을 보내고 응답을 수신.

5. Examples of IPC Systems

Windows의 로컬 프로시저 호출





5. Examples of IPC Systems

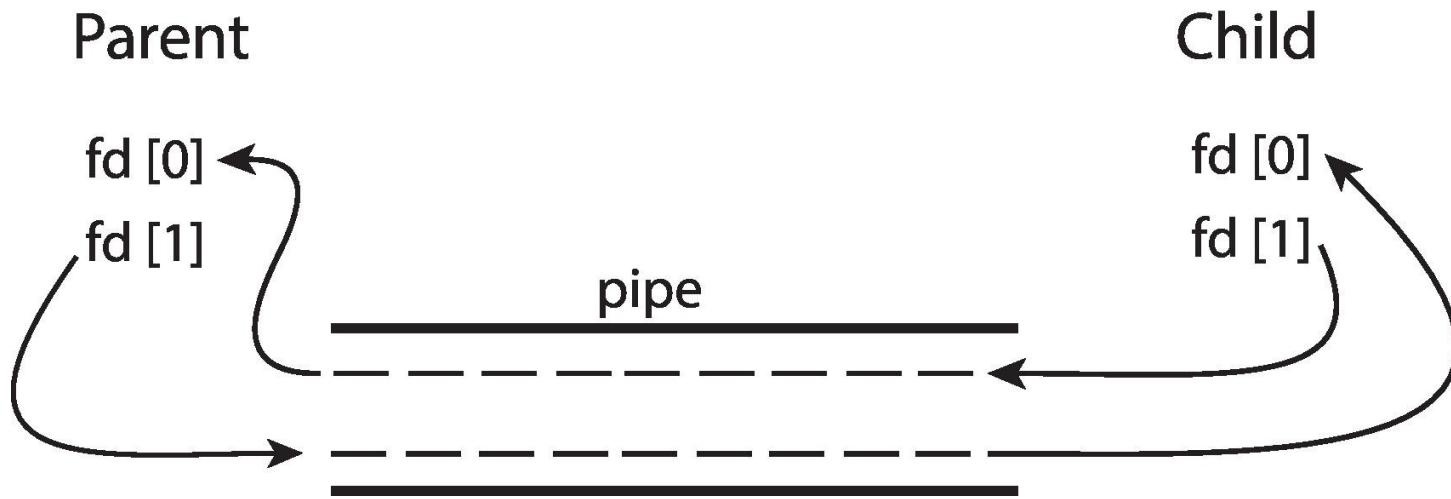
Pipes-Message Passing Examples

- 두 프로세스가 통신할 수 있도록 하는 도관 역할을 한다.
- Issues :
 - 통신은 단방향인가 양방향인가?
 - 양방향 통신의 경우 반이중인가요, 전이중인가요?
 - 통신 프로세스 간에 관계(즉, 부모-자식)가 있어야 합니까?
 - 네트워크를 통해 파이프를 사용할 수 있습니까?
- 일반 파이프 Ordinary pipes – 파이프를 만든 프로세스 외부에서 액세스할 수 없다. 일반적으로 부모 프로세스는 파이프를 만들고 이를 사용하여 자신이 만든 자식 프로세스와 통신.
- 명명된 파이프 Named pipes – 부모-자식 관계 없이 액세스할 수 있다.

5. Examples of IPC Systems

Ordinary Pipes

- 일반 파이프는 표준 생산자-소비자 스타일의 통신을 허용.
- 생산자는 한쪽 끝(파이프의 쓰기 끝)에 쓴다.
- 소비자는 다른 끝(파이프의 읽기 끝)에서 읽는다.
- 따라서 일반 파이프는 단방향



- 통신 프로세스 간에 부모-자식 관계 필요



5. Examples of IPC Systems

Named Pipes

- 명명된 파이프는 일반 파이프보다 강력.
- 통신은 양방향
- 통신 프로세스 간에 부모-자식 관계가 필요하지 않다.
- 여러 프로세스가 통신을 위해 명명된 파이프를 사용할 수 있다.
- UNIX 및 Windows 시스템 모두에서 제공

5. Examples of IPC Systems

UNIX 시스템에서 익명 파이프를 구현

파일

소스코드

실습환경

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > ordinary_pipe.c
```

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main()
{
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    pid_t pid;

    // pipe 생성
    pipe(fd);

    pid = fork(); // 새로운 프로세스 생성

    if(pid>0) // 부모 프로세스(parent process)
    {
        close(fd[READ_END]); // parent는 read할 필요없기 때문에 해제

        // pipe에 메시지 작성
        write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
        close(fd[WRITE_END]);
    }
    else if(pid==0) // 자식 프로세스(child process)
    {
        close(fd[WRITE_END]);

        // pipe 메시지 읽기
        read(fd[READ_END], read_msg, BUFFER_SIZE);
        printf("read %s\n", read_msg);
        close(fd[READ_END]);
    }

    return 0;
}
```

결과값

비고

pipe 함수는 두 개의 정수로 구성된 배열을 가져와서 단방향 파이프를 설정하고, 파이프의 읽기 및 쓰기 끝에 대한 파일 디스크립터를 fd[0] 및 fd[1]에 할당

pid가 0보다 크면 부모 프로세스가 실행 중입니다.

부모 프로세스는 읽기 끝을 닫습니다
(close(fd[READ_END])) 파이프에 쓰기만 하면 되기 때문입니다.

write_msg 메시지를 파이프의 쓰기 끝에 씁니다
(write(fd[WRITE_END], write_msg, strlen(write_msg)+1)). strlen(write_msg)+1은 문자열의 널 종료 문자도 파이프에 쓰기 위한 것입니다.

마지막으로, 부모 프로세스는 쓰기 끝을 닫고 종료합니다(close(fd[WRITE_END])).



5. Examples of IPC Systems

UNIX 시스템에서 익명 파이프를 구현

파일

소스코드

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ gcc ordinary_pipe.c -lrt -o ordinary_pipe.out
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ ./ordinary_pipe.out
```

결과값1

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ read Greetings
```

```
^C
```

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$
```

- **UNIX 시스템에서 익명 파이프를 구현**

비고



5. Examples of IPC Systems

클라이언트-서버 시스템의 통신

- 소켓 Sockets
- 원격 절차 호출 Remote Procedure Calls

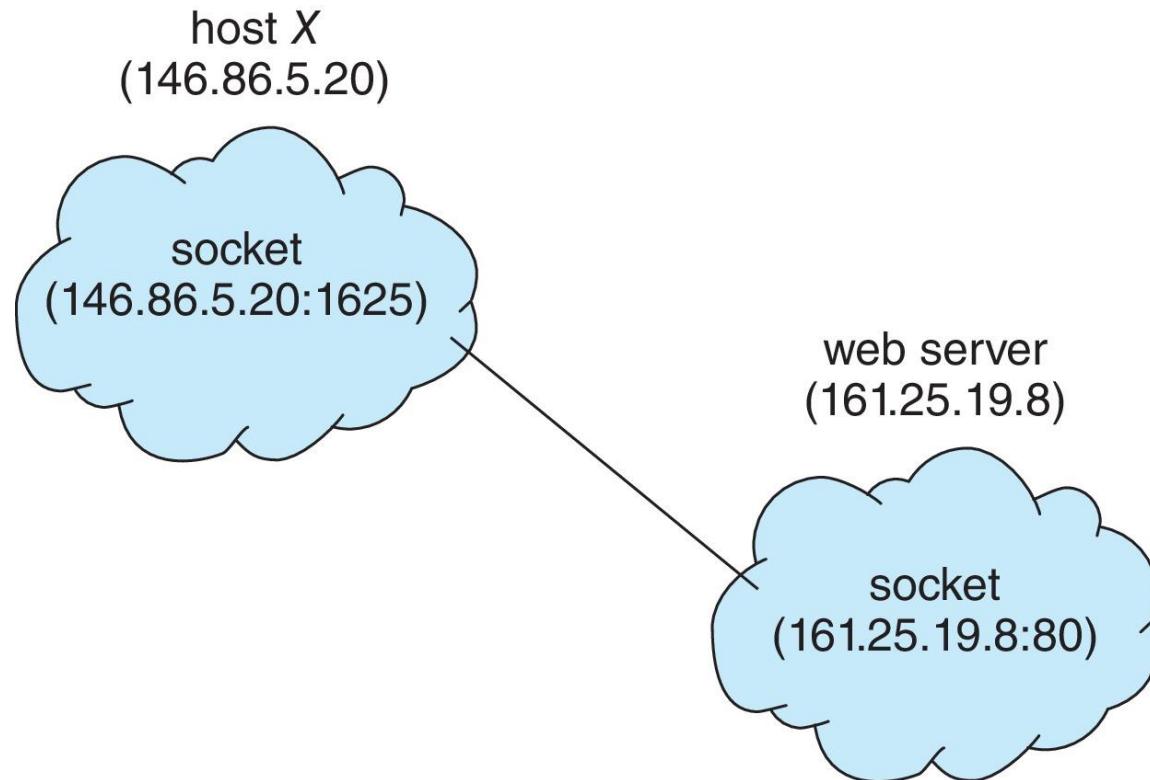
Sockets

- 소켓은 통신을 위한 끝점으로 정의.
- IP 주소와 포트 port 의 연결 - 호스트의 네트워크 서비스를 구별하기 위해 메시지 패킷 시작 부분에 포함된 번호
- 소켓 161.25.19.8:1625는 호스트 161.25.19.8의 포트 1625를 나타낸다.
- 통신은 한 쌍의 소켓으로 구성.
- 1024 미만의 모든 포트는 잘 알려져 있으며 *well known* 표준 서비스에 사용.
- 프로세스가 실행 중인 시스템을 가리키는 특수 IP 주소 127.0.0.1(루프백 loopback)

5. Examples of IPC Systems

클라이언트-서버 시스템의 통신

- **Socket Communication**





5. Examples of IPC Systems

Sockets in Java

- 세 가지 유형의 소켓

- 연결 지향 Connection-oriented(TCP)
- 무접속 Connectionless(UDP)
- MulticastSocket 클래스 – 데이터를 여러 수신자에게 보낼 수 있다.



5. Examples of IPC Systems

Sockets in Java

k8s@DESKTOP-RoEQ2U6:~/c-workspaces\$ javac -version

명령어 'javac' 을(를) 찾을 수 없습니다. 그러나 다음을 통해 설치할 수 있습니다:

sudo apt install openjdk-11-jdk-headless # version 11.0.18+10-oubuntu1~22.04, or

sudo apt install default-jdk # version 2:1.11-72build2

sudo apt install ecj # version 3.16.0-1

sudo apt install openjdk-17-jdk-headless # version 17.0.6+10-oubuntu1~22.04

sudo apt install openjdk-18-jdk-headless # version 18.0.2+9-2~22.04

sudo apt install openjdk-19-jdk-headless # version 19.0.2+7-oubuntu3~22.04

sudo apt install openjdk-8-jdk-headless # version 8u362-ga-oubuntu1~22.04

k8s@DESKTOP-RoEQ2U6:~/c-workspaces\$



5. Examples of IPC Systems

Sockets in Java

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ sudo apt install openjdk-19-jdk-headless
```

[sudo] k8s 암호:

패키지 목록을 읽는 중입니다... 완료

의존성 트리를 만드는 중입니다... 완료

상태 정보를 읽는 중입니다... 완료

...

o added, o removed; done.

Running hooks in /etc/ca-certificates/update.d...

done.

done.

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$
```



5. Examples of IPC Systems

Sockets in Java

파일

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > DateServer.java
import java.net.*;
import java.io.*;

public class DateServer{
    public static void main(String[] args) throws Exception{
        ServerSocket server = new ServerSocket(6013);

        // now listen for connections
        while(true)
        {
            Socket client = server.accept();
            PrintWriter pout = new PrintWriter(client.getOutputStream(), true);

            // socket에 Date 정보 작성
            pout.println(new java.util.Date().toString());

            // client socket 해제
            client.close();
        }
    }
}
```

소스코드

결과값1

비고

- 1024 미만의 포트를 사용 시 "관리자 권한" 포트로 선언해야 함
- [java.net.BindException](#)



5. Examples of IPC Systems

Sockets in Java

파일

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c-workspaces$ cat > DateClient.java
import java.net.*;
import java.io.*;

public class DateClient{
    public static void main(String[] args) throws Exception{
        // make connection to server socket
        Socket socket = new Socket("127.0.0.1", 6013);

        InputStream in = socket.getInputStream();
        BufferedReader br = new BufferedReader(new InputStreamReader(in));

        // read date from the socket
        String line = null;
        while((line=br.readLine())!=null)
        {
            System.out.println(line);
        }

        // close the socket connections
        socket.close();
    }
}
```

소스코드

결과값¹

비고

- 1024 미만의 포트를 사용 시 "관리자 권한" 포트로 선언해야 함
- [java.net.BindException](#)



5. Examples of IPC Systems

Sockets in Java

파일

소스코드

실습환경

terminal 1

소스코드

k8s@DESKTOP-RoEQ2U6:~/c-workspaces\$ javac DateServer.java

결과값1

k8s@DESKTOP-RoEQ2U6:~/c-workspaces\$ javac DateClient.java

비고

파일

소스코드

실습환경

terminal 1

소스코드

k8s@DESKTOP-RoEQ2U6:~/c-workspaces\$ java DateServer

결과값1

비고

파일

소스코드

실습환경

terminal 2

소스코드

k8s@DESKTOP-RoEQ2U6:~/c-workspaces\$ java DateClient

결과값1

Wed May 03 18:25:47 KST 2023

k8s@DESKTOP-RoEQ2U6:~/c-workspaces\$

비고



5. Examples of IPC Systems

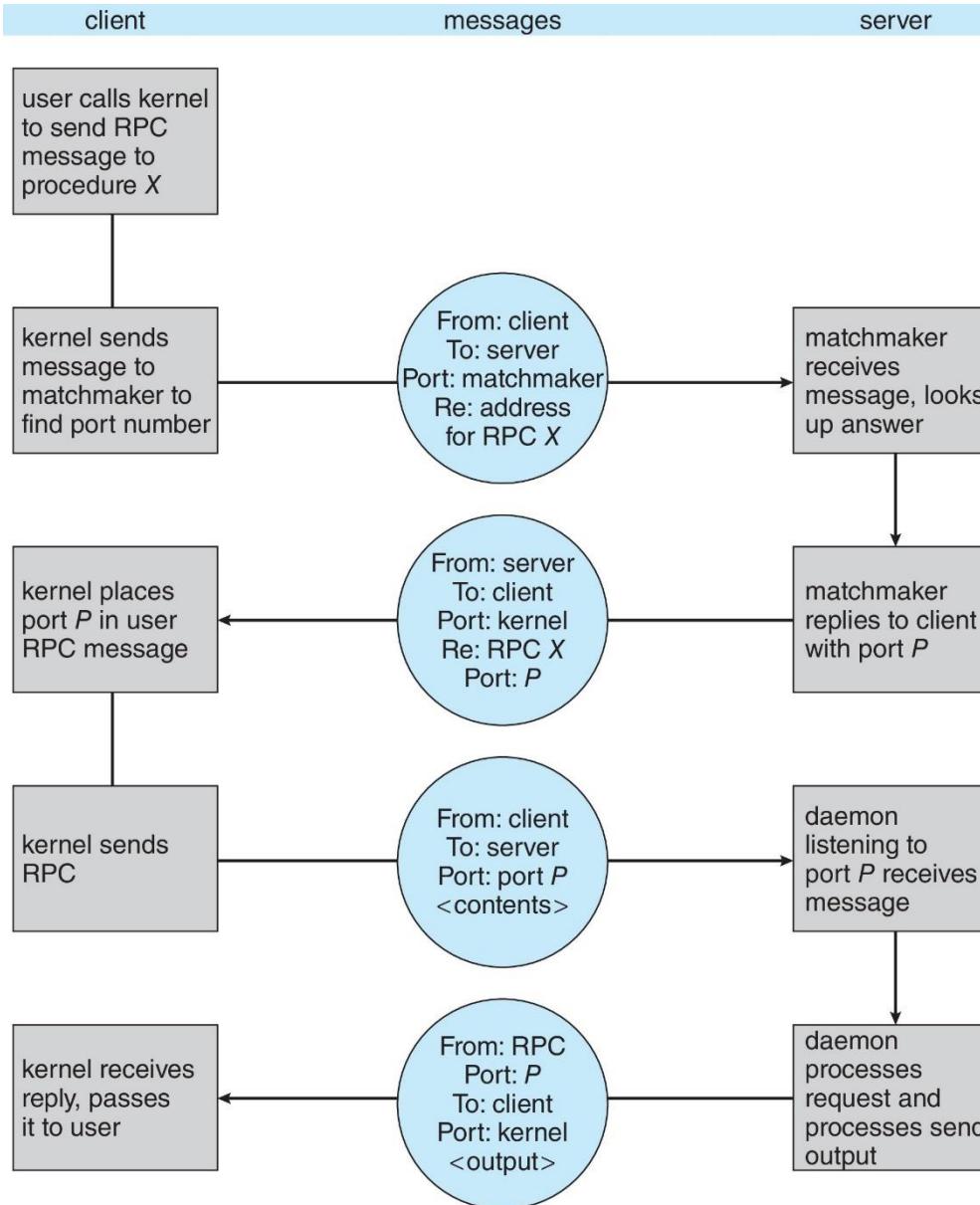
Remote Procedure Calls

- RPC(원격 프로시저 호출)는 네트워크 시스템의 프로세스 간 프로시저 호출을 추상화.
 - 다시 서비스 차별화를 위해 포트 사용
- 스텁 Stubs - 서버의 실제 프로시저에 대한 클라이언트 측 프록시
 - 클라이언트 측 스텁은 서버를 찾고 매개변수를 마샬링 marshalls .
 - 서버측 스텁은 이 메시지를 수신하고 마샬링된 매개변수의 압축을 풀고 서버에서 절차를 수행.
- Windows에서 MIDL(Microsoft Interface Definition Language)로 작성된 사양에서 스텁 코드 컴파일
- 다양한 아키텍처를 설명하기 위해 XDL(External Data Representation) 형식을 통해 처리되는 데이터 표현
 - 빅엔디안 Big-endian 과 리틀엔디안 little-endian
- 원격 통신에는 로컬보다 더 많은 실패 시나리오가 있다.
- 메시지는 최대 한 번이 아니라 정확히 한 번 배달될 수 있다.
- OS는 일반적으로 클라이언트와 서버를 연결하는 랑데뷰(또는 매치메이커 matchmaker) 서비스를 제공.

5. Examples of IPC Systems



RPC 실행





6. 파일을 이용한 통신 Examples

파일

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ cat > filetest.c
```

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

```
int main()
```

```
{
```

```
    FILE *fp;
```

```
    char buf[5];
```

```
    fp = fopen("filetest.txt", "w+"); // 파일을 쓰기 및 읽기 모드로 열기
```

```
    fwrite("test", sizeof(char), 4, fp); // 파일에 데이터 쓰기
```

```
    fseek(fp, 0, SEEK_SET); // 파일 포인터를 파일의 처음으로 이동
```

```
    fread(buf, sizeof(char), 4, fp); // 파일에서 데이터 읽기
```

```
    buf[4] = '\0'; // 문자열 종료를 위해 널 문자 추가
```

```
    printf("%s\n", buf); // 읽은 데이터 출력
```

```
    fclose(fp); // 파일 닫기
```

```
    return 0;
```

```
}
```

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ gcc filetest.c
```

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ ls
```

```
Sample.h fork_ex1.c fun2.c gdb_test.c helloworld.o main.o      omp_2.c      process_011.c  
a.out   fun1.c   fun2.o hello     main     memory_layout.c process_001.c pthread.c  
filetest.c fun1.o   gdb_test helloworld.c main.c      omp_1.c      process_007.c
```

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ ./a.out
```

```
test
```

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ cat filetest.txt
```

```
testk8s@DESKTOP-RoEQ2U6:~/c_workspaces$
```





참고

프로세스 제어 블록(process control block, PCB)

Process-Id
Process state
Process Priority
Accounting Information
Program Counter
CPU Register
PCB Pointers
.....

Process Control Block



참고

프로세스 제어 블록(process control blcok, PCB)

1. process-id : 새로운 프로세스에 시스템이 할당해주는 고유 id
2. process-state : 프로세스의 라이프 타임과 관련된 상태로, waiting, running, ready, blocked, end, suspend-wait, suspend-ready 가 있다.
3. priority : 프로세스 스케줄링을 위한 우선순위이다.
4. Process Accounting Information : CPU를 사용한 시간, CPU 할당 시간 등이 있다.
5. Program Counter : 이전에 배운 PC로 다음 작업할 명령어 위치를 기억한다.
6. List of Open Files : 실행 중에 프로그램에 필요한 모든 파일의 정보를 포함한다.
7. Process I/O status information : 해당 프로세스가 실행 중에 할당을 요구한 I/O 장치에 대한 정보를 담는다.
8. CPU 레지스터 : context switch가 발생하면 이 때의 레지스터 정보를 기억해서 다시 프로세스가 CPU 할당을 받으면 사용한다. accumulator, index, stack pointer 와 같은 레지스터의 값이 저장된다.
9. PCB Pointer : 준비중인 다음 프로세스의 주소를 가리킨다. 준비 상태나, 대기 상태의 큐를 구현할 때 다음을 가리키는 포인터로 사용된다.
10. Memory management information : 메모리 관리 정보로 프로세스가 메모리의 어디에 있는지 나타내는 메모리 정보와 메모리 보호를 위한 경계 레지스터, 한계 레지스터 값등이 저장된다. 또한, segmentation table , page table 등 정보도 보관한다.
11. PPID, CPID : 부모 프로세스를 가리키는 PPID, 자식 프로세스를 관리하는 CPID가 저장된다.



빅엔디안 Big-endian 과 리틀엔디안little-endian

- **빅엔디안 (Big-endian)**: 가장 상위 바이트부터 메모리에 저장하는 방식입니다. 즉, 가장 큰 단위의 바이트가 가장 낮은 메모리 주소에 위치합니다. 네트워크에서는 보통 빅엔디안을 사용합니다.
- **리틀엔디안 (Little-endian)**: 가장 하위 바이트부터 메모리에 저장하는 방식입니다. 즉, 가장 작은 단위의 바이트가 가장 낮은 메모리 주소에 위치합니다. 인텔과 같은 x86 아키텍처에서 주로 사용됩니다.
- 컴퓨터 아키텍처에 따라 엔디안 방식이 다를 수 있으며, 대부분의 시스템은 빅엔디안 또는 리틀엔디안 중 하나를 따릅니다. 예를 들어, x86 아키텍처는 리틀엔디안을 사용하고, SPARC 및 IBM POWER 아키텍처는 빅엔디안을 사용합니다.
- 프로그래밍에서는 엔디안에 대한 고려가 필요한 경우가 있습니다. 특히 네트워크 통신이나 이식성이 있는 데이터 저장 및 교환에 관련됩니다. C나 C++에서는 <netinet/in.h> 헤더 파일의 htonl, htons, ntohs, ntohl, ntohs와 같은 함수를 사용하여 호스트 바이트 순서와 네트워크 바이트 순서 사이의 변환을 수행할 수 있습니다.



랑데뷰(또는 매치메이커 matchmaker) 서비스

- 랑데뷰(또는 매치메이커) 서비스는 사람들을 서로 매칭시켜주는 온라인 플랫폼 또는 애플리케이션을 말합니다. 이러한 서비스는 주로 다음과 같은 목적을 가지고 운영됩니다
 1. 데이트 매칭: 랑데뷰 서비스는 개인의 관심사, 성향, 취향 등을 기반으로 사용자들을 매칭시켜주어 데이트 또는 로맨틱한 관계를 형성할 수 있도록 돕습니다. 사용자들은 프로필을 작성하고 상대방의 프로필을 검토하여 자신과 잘 맞는 상대를 찾을 수 있습니다.
 2. 친구 찾기: 일부 랑데뷰 서비스는 로맨틱한 관계뿐만 아니라 새로운 친구를 찾을 수 있는 기능을 제공하기도 합니다. 사용자들은 공통의 관심사나 활동을 가진 사람들을 찾아 함께 시간을 보내거나 친분을 쌓을 수 있습니다.
 3. 전문 분야의 매칭: 일부 랑데뷰 서비스는 전문 분야나 업무 관련하여 사람들을 매칭시켜주는 기능을 제공합니다. 예를 들어, 멘토-멘티 관계 형성, 비즈니스 파트너 찾기, 공동 연구를 위한 파트너 찾기 등이 있습니다.
- 랑데뷰 서비스는 사용자들의 프로필 정보, 취향, 선호도 등을 기반으로 매칭 알고리즘을 사용하여 가장 적합한 상대를 찾아주는 역할을 합니다. 많은 랑데뷰 서비스는 사용자들에게 편리한 기능과 검색 옵션, 커뮤니케이션 도구 등을 제공하여 사용자 경험을 향상시키고 효과적인 매칭을 지원합니다.
- 주요 랑데뷰 서비스에는 틴더(Tinder), 매치(Match), 오키드(Coffee Meets Bagel), 바디프렌드(Bumble) 등이 있습니다.



참고

마샬링 marshalls

- 마샬링(Marshalling)은 컴퓨터 과학에서 데이터를 한 시스템에서 다른 시스템으로 전송하거나 저장하기 위해 데이터를 일련의 바이트 스트림으로 변환하는 과정을 말합니다. 이러한 변환은 데이터를 네트워크 상에서 전송하거나 디스크에 저장하고 나중에 다시 읽거나 사용할 수 있도록 합니다.
- 마샬링은 다른 컴퓨터 아키텍처나 다른 프로그래밍 언어 간의 데이터 교환을 가능하게 합니다. 각 시스템 또는 프로그래밍 언어는 데이터를 표현하는 방식이 다를 수 있으며, 이는 데이터의 바이트 순서, 정수와 실수의 표현 방식, 문자열 인코딩 등에서 나타납니다. 마샬링은 이러한 차이점을 극복하고 데이터를 표준화된 형식으로 변환하여 전송 및 저장할 수 있도록 해줍니다.
- 마샬링은 주로 네트워크 프로그래밍, 원격 프로시저 호출(RPC), 분산 시스템 등에서 사용됩니다. 데이터를 마샬링하는 과정에서는 데이터를 직렬화하여 바이트 스트림으로 변환하고, 수신 측에서는 이를 역직렬화하여 원래의 데이터 형식으로 복원합니다.
- 많은 프로그래밍 언어 및 프레임워크에서 마샬링을 지원하는 라이브러리나 기능을 제공합니다. 예를 들어, 자바에서는 직렬화(Serialization)와 관련된 클래스와 인터페이스를 제공하며, C#에서는 .NET의 직렬화 기능을 활용할 수 있습니다. 또한, 웹 서비스와 같은 분산 시스템에서는 XML 또는 JSON과 같은 데이터 포맷을 사용하여 마샬링을 수행합니다.