**시스템 프로그래밍(SW) 과제 1**

**학번: 32193430**

**이름: 이재원**

**담당교수: 최 종 무 교수님**

**분반: 3분반**

**제출일: 2021. 09. 17**

**I. Summary**

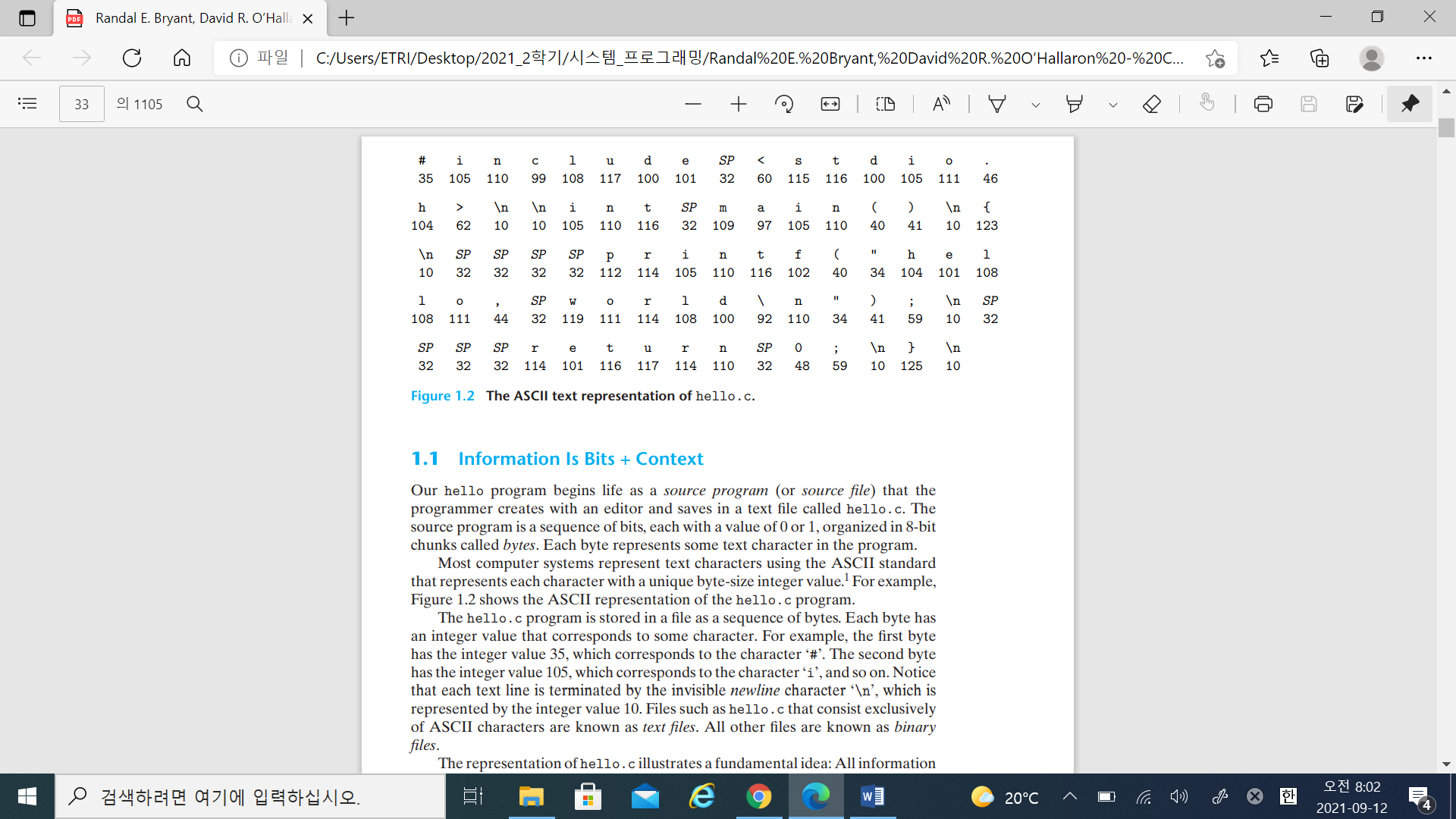
**A Tour of Computer Systems**

컴퓨터 시스템 둘러보기

이 장에서는 간단한 “hello world” 프로그램의 생명 주기를 추적함으로써, 컴퓨터 시스템의 주요한 아이디어와 주제를 소개한다.

* 1. **Information Is Bits + Context**

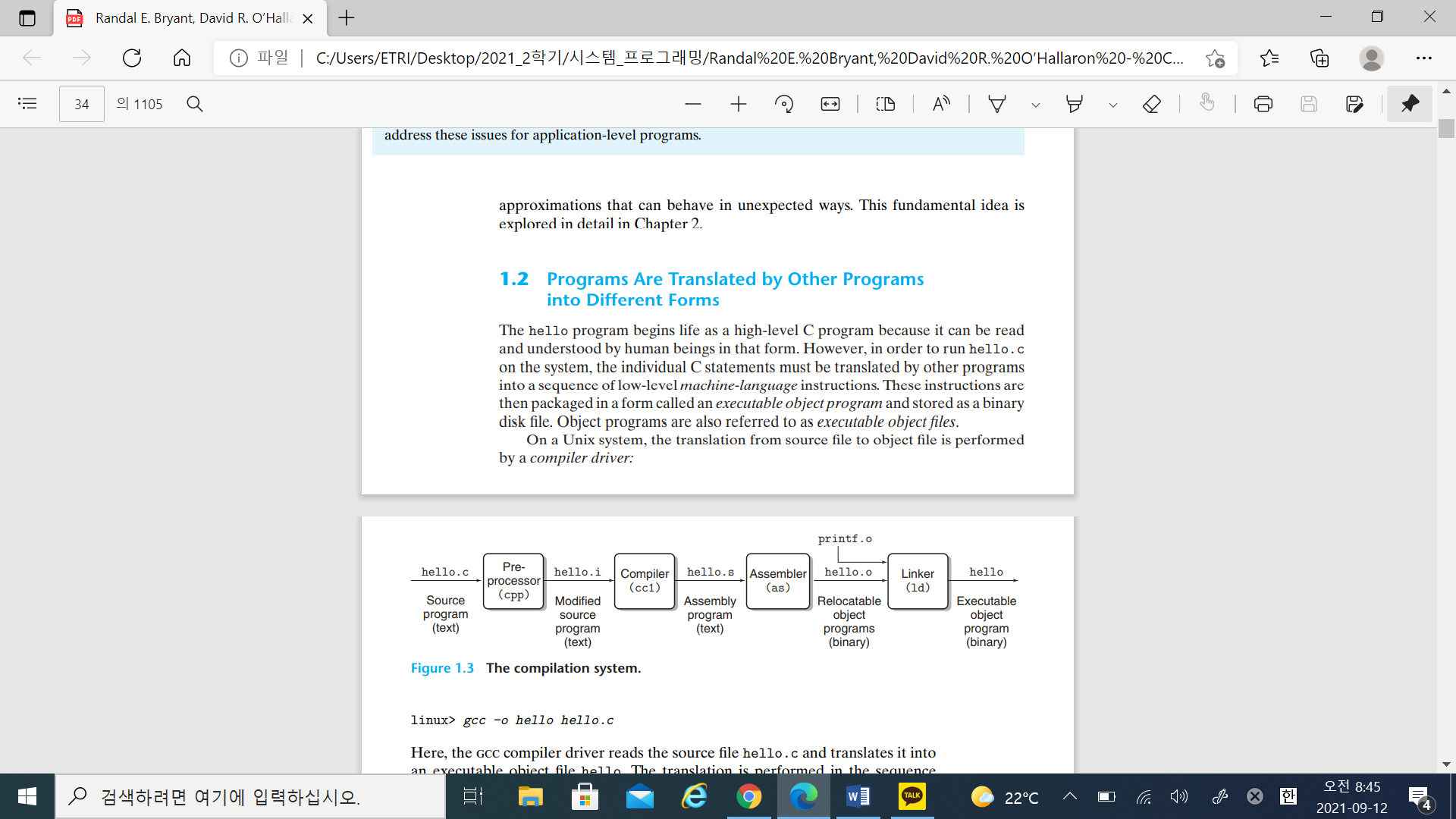
대부분의 컴퓨터 시스템은 고유한 byte 크기의 정수 값으로 각 문자를 나타내는 ASCII 표준을 사용하여 텍스트 문자를 나타낸다.



디스크 파일, 메모리에 저장된 프로그램과 사용자 데이터 및 네트워크를 통해 전송되는 데이터를 포함한 시스템의 모든 정보는 여러 개의 bits로 표현된다. 각 데이터가 데이터 개체가 서로 다르다는 것은 오직 우리가 보는 context로만 구별할 수 있다. **정보는 Bits와 Context의 합이다**.

* 1. **Programs Are Translated by Other Programs into Different Forms**

Hello 프로그램은 인간에 의해 읽고 이해될 수 있는 high-level C 언어로 시작한다. 그러나 시스템에서 hello.c를 실행하기 위해서는 각각의 C 문장이 다른 프로그램에 의해 낮은 수준의 machine-language 명령으로 변경되어야 한다. 이 명령어들은 executable object program이라고 불리는 형태로 포장되어 binary로 저장된다.



여기서 gcc compiler driver는 소스 파일 hello.c를 읽고 실행 가능한 오브젝트 파일 hello로 번역한다. 번역은 그림 1.3에 표시된 네 단계의 순서로 수행된다. 4단계(preprocessor, compiler, assembler, and linker)를 수행하는 프로그램들의 집합을 compilation system이라고 한다. **프로그램은 다른 형태의 다른 프로그램으로 번역된다.**

* 1. **It Pays to Understand How Compilation Systems Work**

**프로그래머들이 compilation system이 어떻게 작동하는지 이해해야 하는 3가지 이유**

1. Optimizing program performance(프로그램 성능을 최적화하기 위해서)

현대의 compiler는 좋은 코드를 생성하기 위해 정교한 작업을 수행할 수 있는 도구이다. 프로그래머들은 효율적인 코드를 작성하기 위해 컴파일러의 내부 작업까지는 굳이 알 필요가 없다. 그러나 우리가 machine-level code에 대한 기본적인 이해를 하고 있고, compiler가 C언어 문장을 machine code로 변환하는 방법을 알면 코딩을 할 때 좋은 결정을 내릴 수 있다.

1. Understanding link-time errors(링크 시간 오류를 이해하기 위해서)

linker의 작동과 관련하여 가장 복잡한 프로그래밍 오류가 발생하기도 한다. 이는 특히 거대한 소프트웨어를 만들려고 할 때 자주 관찰할 수 있다.

Ex) linker가 참조를 해결할 수 없다고 보고하는 것은 무슨 의미인가? 정적 변수와 전역 변수의 차이점은 무엇일까? 동일한 이름의 서로 다른 C 파일에 두 개의 전역 변수를 정의하면 어떻게 될까? **왜 linker 관련 오류가 실행 시간까지 나타나지 않는가?**

1. Avoiding security holes(보안의 허점을 방지하기 위해서)

buffer overflow 취약성은 네트워크 및 인터넷 서버의 많은 보안 결함을 설명했다. 신뢰할 수 없는 출처에서 수락하는 데이터의 양과 형식을 신중하게 제한해야 할 필요성을 이해하는 프로그래머가 너무 적기 때문에 이러한 취약성이 발생한다. 보안 프로그래밍을 배우는 첫 번째 단계는 데이터 및 제어 정보가 프로그램 stack에 저장되는 방법과 그 결과를 이해하는 것이다.

**프로그래머들이 compilation system이 어떻게 작동하는지 이해하는 것은 도움이 된다.**

* 1. **Processors Read and Interpret Instructions Stored in Memory**

Processor는 메모리에 저장된 명령을 읽고 해석한다.

* + 1. **Hardware Organization of a System**

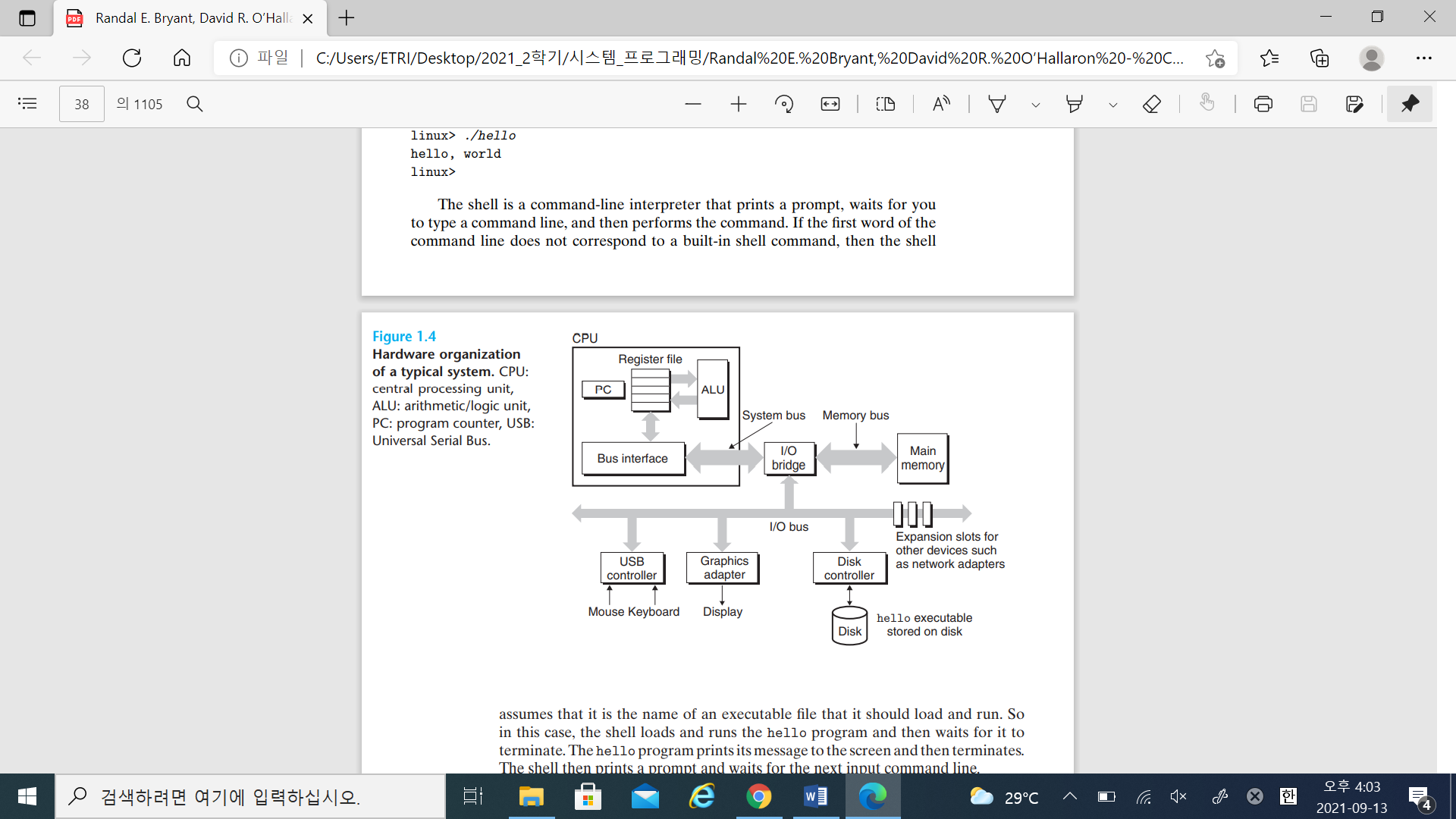


Figure 1.4 일반적인 시스템의 하드웨어 구성

- Buses

Buses라고 불리는 전기 전선관의 집합이 바이트의 정보를 전달하면서 시스템 전체가 실행된다. Buses는 일반적으로 word로 알려진 고정된 크기의 바이트 덩어리를 전송하도록 설계되어 있다. Word 안에서 바이트 수는 시스템에 따라 달라지는 기본 시스템 매개변수이다. 오늘날 대부분의 컴퓨터는 word 크기가 32bit이거나 64bit이다.

- I/O Devices

Input/output (I/O) 장치는 시스템을 외부 세계에 연결하는 장치이다.

Ex) 사용자 입력을 위한 키보드와 마우스, 사용자 출력을 위한 디스플레이, 데이터 및 프로그램의 장기 저장을 위한 디스크 드라이브(또는 단순히 디스크) 등 4개의 I/O 장치가 있다.

- Main Memory

Main memory는 processor가 프로그램을 실행하는 동안 프로그램과 데이터를 모두 저장하는 임시 저장 장치이다. 물리적으로 Main Memory는 DRAM 칩의 집합으로 구성되어 있다.

- Processor

중앙 처리 장치(CPU) 또는 일명, processor는 main memory에 저장된 명령을 해석하거나 실행하는 엔진이다. 그 중심에는 Program Counter(PC)라고 불리는 word 사이즈의 저장 장치(또는 register)가 있다. 언제나 PC는 Main Memory의 기계어 명령을 가리키고 있다.

CPU가 명령에 따라 수행할 수 있는 간단한 작업의 예

* Load: Main memory의 바이트 또는 단어를 register로 복사하여 register의 이전 내용을 덮어쓴다.
* Store: register에서 main memory의 위치로 바이트 또는 단어를 복사하여 해당 위치의 이전 내용을 덮어쓴다.
* Operate: register 2개의 내용을 ALU에 복사하여 두 단어에 대한 산술 연산을 수행하고 그 결과를 register에 저장하여 이전 내용을 덮어쓴다.
* Jump: 명령에서 단어를 추출하여 해당 단어를 PC에 복사하여 PC의 이전 값을 덮어쓴다.

**1.4.2 Running the hello Program**

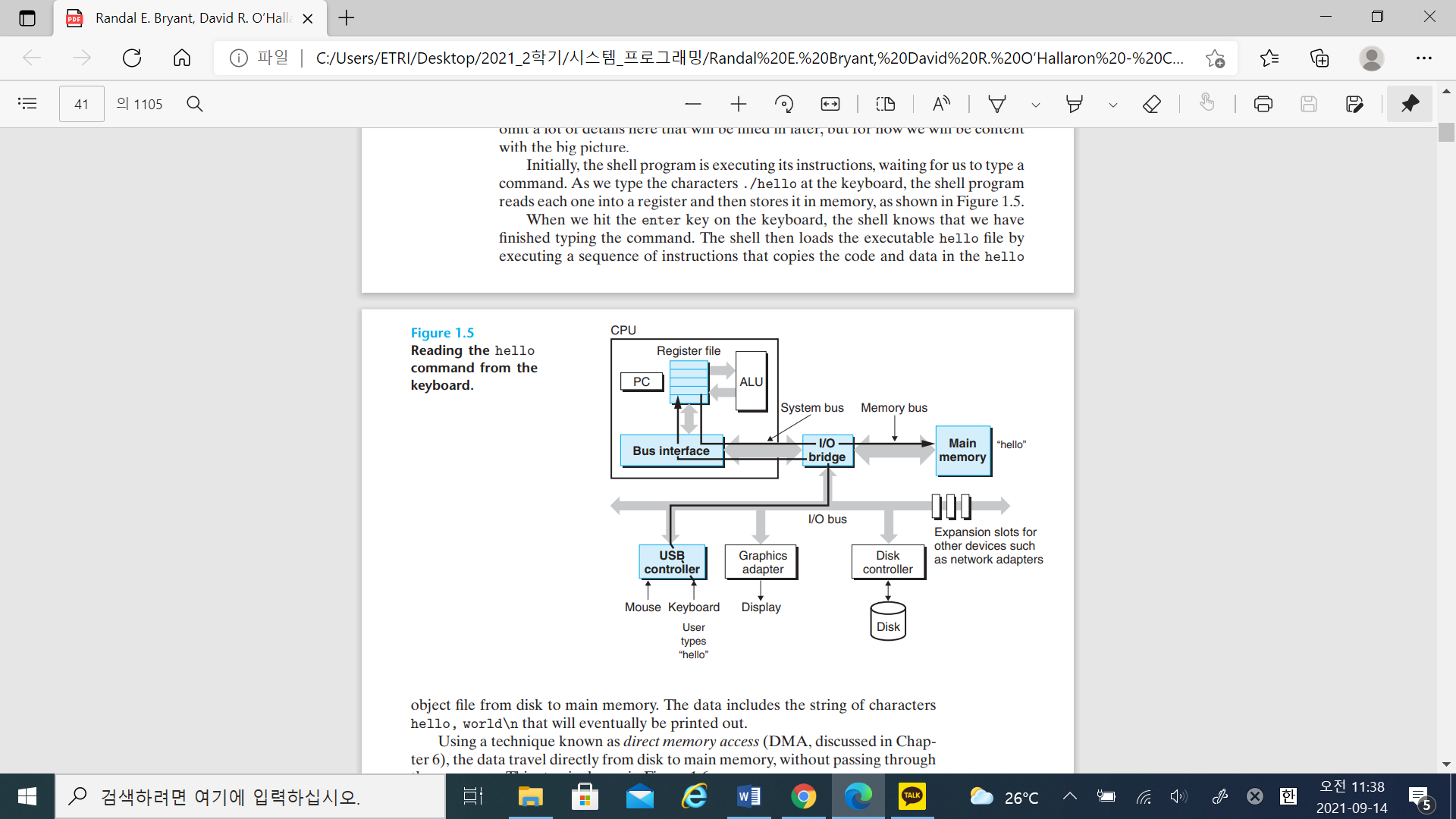


Figure 1.5 키보드로부터 *Hello* 명령 읽기

처음에 shell 프로그램은 명령어를 입력하기 위해 명령을 실행하는 중이다. 키보드에 **./hello** 문자를 입력하면 shell 프로그램이 각각의 register에 읽은 다음 메모리에 저장한다. 키보드의 Enter 키를 누르면 shell이 명령 입력이 끝났음을 알 수 있다. 그런 다음 shell은 디스크로부터 main memory로 hello 객체 파일의 코드와 데이터를 복사하는 명령을 실행하여 실행 파일 hello 파일을 load 한다. 최종적으로 출력될 hello, world\n 문자열이 데이터에 포함된다.

* 1. **Caches Matter**

일반적인 register file은 Main Memory에 수십억 바이트가 아닌 수백 바이트의 정보만 저장하지만, processor는 register 파일에서 데이터를 메모리보다 거의 100배 빠르게 읽는다. 더욱 문제가 되는 것은 반도체 기술이 수년에 걸쳐 발전함에 따라 processor-memory 격차가 계속 증가하고 있다는 점이다. Main memory보다 processor를 더 빨리 구동 시키는 것이 더 쉽고 저렴하다.

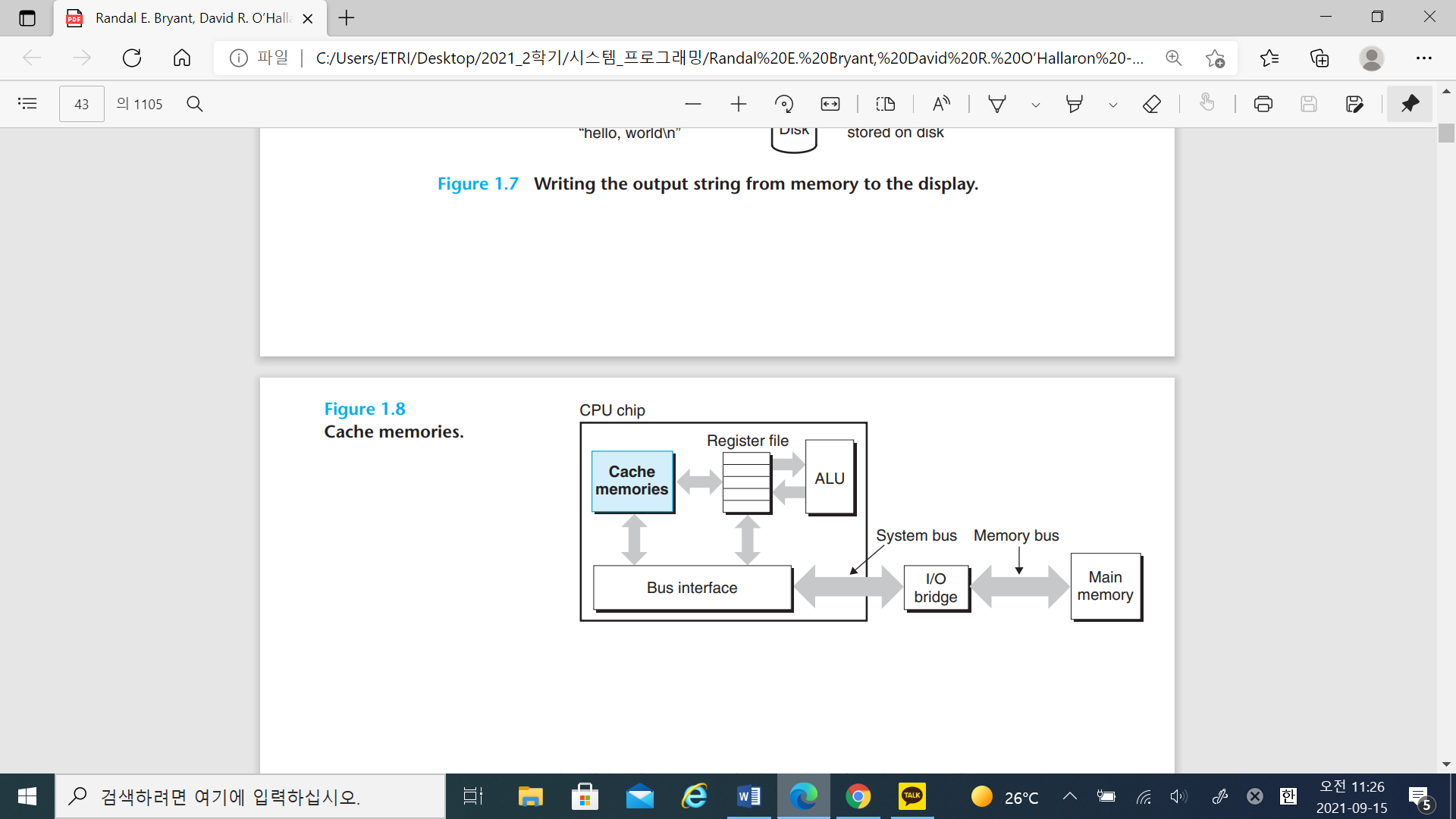


Figure 1.6 Cache Memories

Processor-Memory 차이를 해결하기 위해 시스템 설계자들은 Cashes Memory(또는 단순히 cashes)라고 불리는 더 작고 빠른 Storage 장치를 포함하고 있으며, 이 장치는 Processor가 가까운 미래에 필요로 할 정보를 전달하기 위해 임시로 준비한다.

* 1. **Storage Devices Form a Hierarchy**

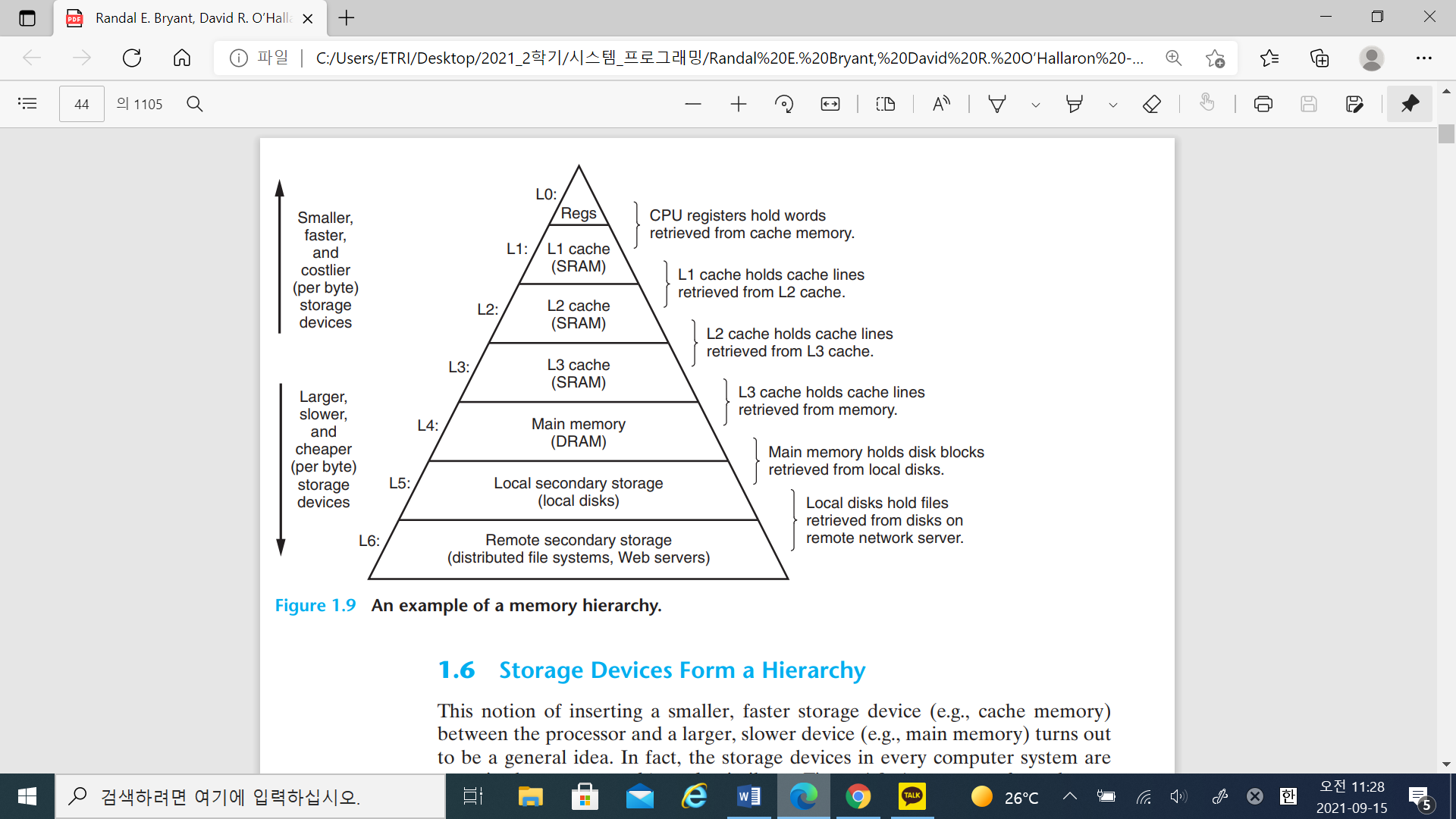


Figure 1.7 Memory Hierarchy의 예시

Hierarchy의 맨 위에서 아래로 이동함에 따라 장치가 느려 지고, 커지고, 바이트 당 비용이 적게 든다.

Memory Hierarchy의 주요 아이디어는 한 레벨의 storage는 위 레벨을 위한 cache 역할을 한다는 것이다.

* 1. **The Operating System Manages the Hardware**

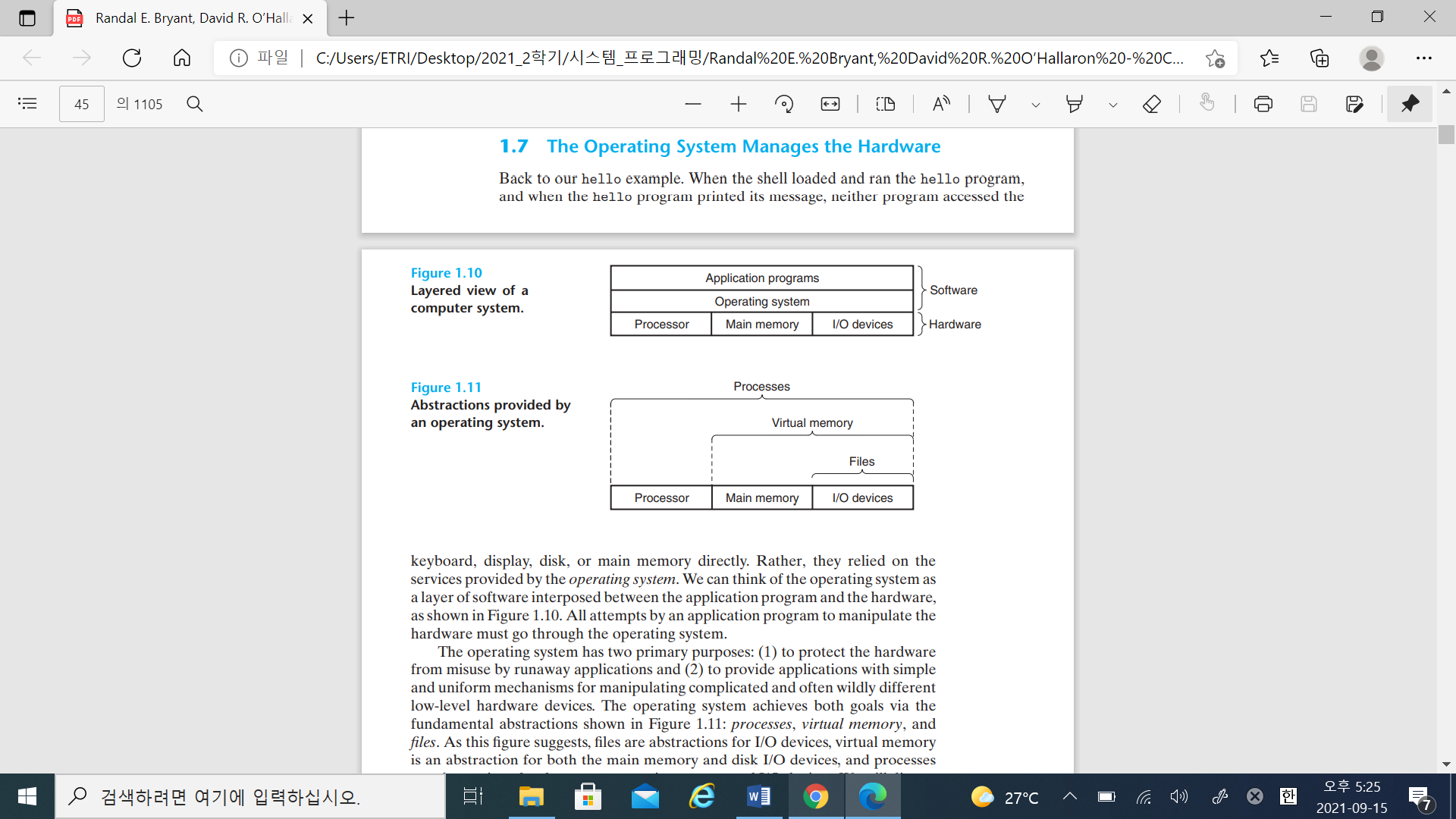


Figure 1.7 컴퓨터 시스템의 단계

Application Program에서 Hardware를 조작하려면 반드시 Operation System을 거쳐야 한다.

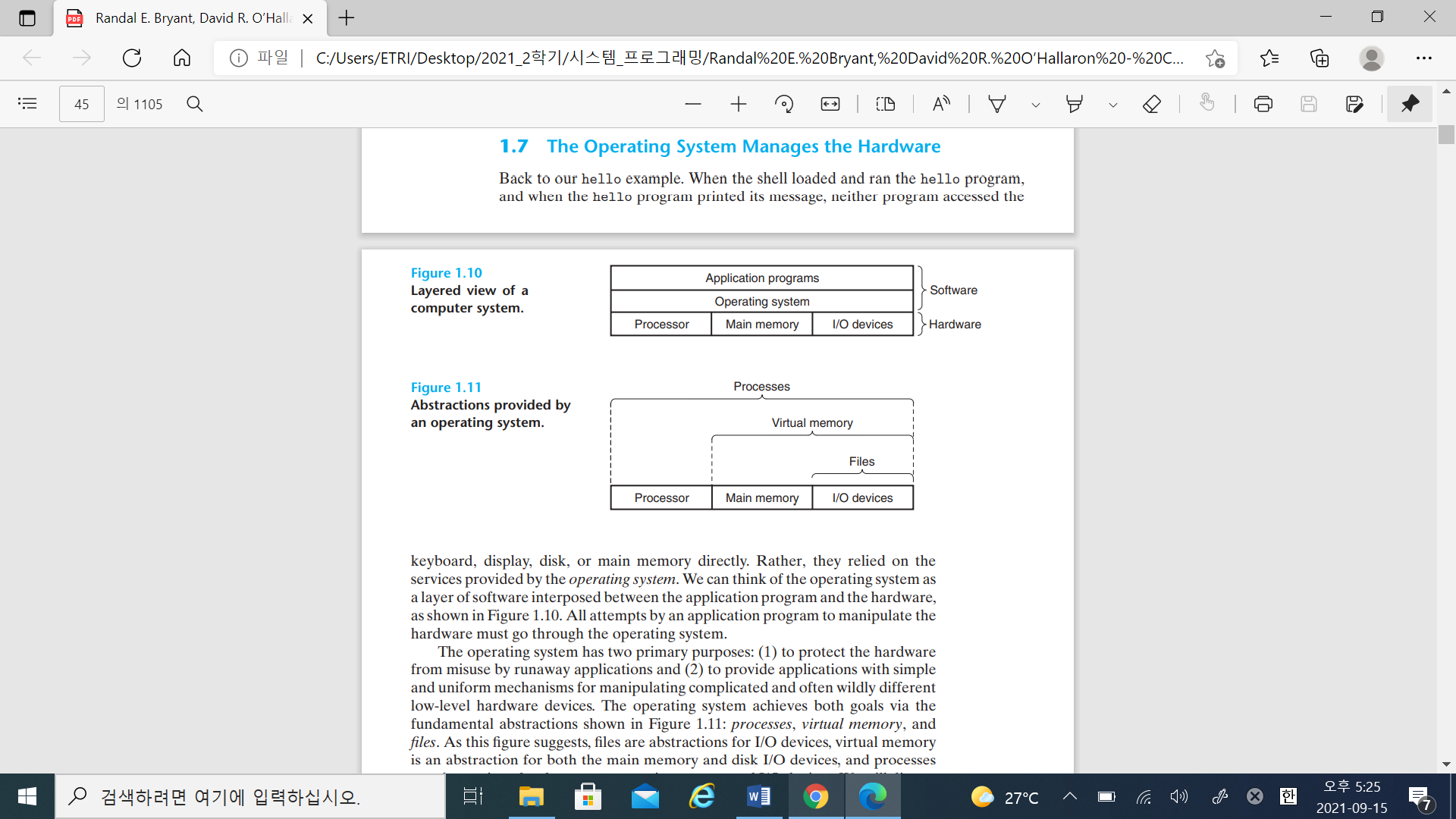


Figure 1.8 OS에서 제공하는 추상화

OS의 주요 목표 2가지

(1) 제어가 안 되는 애플리케이션에 의해 hardware가 잘못 사용되는 것을 보호한다.

(2) 복잡하고 많이 다른 low-level hardware 장치를 조작하기 위한 간단하고 적절한 메커니즘을 애플리케이션에 제공한다.

OS는 Figure 1.8에 표시된 기본적인 추상화, 즉 프로세스, 가상 메모리 및 파일을 통해 두 가지 목표를 모두 달성한다.

**II. Purpose of studying System Programming**

1. **교과목에서**

시스템 프로그래밍 교과목의 목표는 “소프트웨어가 하드웨어에서 어떻게 실행되는지(소프트웨어와 하드웨어가 어떻게 연결되는지)”를 이해하는 것 그리고 abstraction(추상화)의 개념을 이해하는 것이다. 이를 위해 다음과 같은 개념들을 공부할 것이다.

**소프트웨어가 하드웨어에서 어떻게 실행되는가?**

1. 인간을 위한 high-level 프로그램 vs CPU를 위한 이진법
2. Compiler, Assembler, Linker, Loader, Debugger, Library
3. File system과 Device driver
4. Process의 개념, Scheduling for multiple processes
5. Memory 관리 (data/stack/heap, virtual memory(가상 메모리))
6. Software-level optimizations(최적화): code motion, loop unrolling, …
7. Hardware-level optimizations: pipeline, cache, …
8. Recent technologies in Intel CPU

**abstraction(추상화)**

1. Information hiding
2. Interface vs. Implementation
3. Layered architecture
4. **궁극적인 질문**

저번 여름방학 때부터 단국인재아카데미라는 프로그램에 참여하고 있다. 지정된 도서에 대해 학생과 교수가 공동으로 수업하고, 이후 다양한 문제에 관해 토론하는 식으로 진행한다. 필자는 서울대학교 기초교육원 김광식 교수님과 협업하여 유발 하라리의 <호모데우스: 미래의 역사> 수업과 토론 진행을 맡아 준비하였다. 저자는 이 책에서 약 7만 년 전 ~ 2090년까지의 ‘미래의 역사’ 를 다룬다. “호모 사피엔스는 어디에서 왔으며 앞으로 어디로 갈 것인가? 우리는 생명공학과 **‘컴퓨터 과학’의 힘을 빌려 행복하고, 영원히 죽지 않으며, 세상을 통치할 신성을 획득할 것이다**. 우리는 스스로를 업그레이드하여 신에 필적한 힘을 가진 인간, 호모데우스가 될 것이다”라고 하라리는 주장한다. 책을 읽는 내내 저자의 현대 사회에 대한 날카로운 분석과 미래에 대한 통찰력에 감탄하였다. 필자는 앞으로 시스템 프로그래밍, 더 나아가 평생 컴퓨터 과학을 공부하면서 다음 질문에 대한 나의 대답을 찾을 것이다.

**1. 유기체는 단지 알고리즘이고, 생명은 실제로 데이터 처리 과정에 불과할까?**

**2. 지능과 의식 중에서 무엇이 더 가치 있을까?**

**3. 의식은 없지만, 지능이 매우 높은 알고리즘이 우리보다 우리 자신을 더 잘 알게 되면 사회, 정치, 일상에 어떤 일이 일어날까?**

**4. 나는 무엇을 원하고 싶은가(내 자신의 욕망을 어떻게 설계할까?)**

**5. AI가 대답할 수 없는 질문은 무엇일까(알고리즘이 침범할 수 없는 인간만의 고유한 영역이 있는가?)**

유기체가 단지 알고리즘이라면, 생명체의 의식과 지능을 소프트웨어로 구현할 수 있는가?

1946년 펜실베이니아 대학교에서 최초의 전자 컴퓨터 ENIAC을 개발한 이후로 컴퓨터 과학의 역사를 살펴보면, 컴퓨터의 지능을 놀랄 만큼 발전하였지만, 의식은 전혀 발달하지 않았다. 인간은 의식보다 지능을 중요시하기 때문이다. 예를 들어, 1885년 자동차가 발명되기 이전 말은 주요한 교통수단이었다. 하지만 ‘A 지점에서 B 지점으로 빠르게 이동하는 능력’이라는 관점에서, 자동차의 지능은 말보다 높았다. 말에게는 의식이 있었지만, 자동차에는 없었다. 말은 자동차로 교체되었다. 지능이 의식보다 중요하기 때문이다. 이 흐름을 지속한다면, 완벽한 자율주행 자동차가 나올 것이고 대부분의 운송 노동자들은 실업자가 될 것이다. 지능과 의식 중에서 무엇이 더 가치 있는가?

현대에 사는 우리는 삶의 많은 시간을 알고리즘에 의존한다. 어떤 동영상을 볼까, 어떤 물건을 구매할까, 어떤 이성과 만날까 등 사소해 보이지만 중요한 결정을 구글, 페이스북, 인스타그램 알고리즘에 맡긴다. 이대로 라면, 우리는 점점 더 “어디에서 살 것인가? 어떤 직업을 가질 것인가? 어떤 배우자를 만날 것인가?”와 같은 삶에 있어서 중요한 결정을 대부분 알고리즘에 맡길지도 모른다. 알고리즘이 인간보다 인간을 더 잘 알기 때문이다.

그렇게 매력적인 미래는 아니다. 시스템 프로그래밍을 공부하면서 위에서 제시한 질문들을 어떻게 해결할 것인지 고민할 것이다.

참고문헌: 유발 노아 하라리, <호모데우스>, 김명주 옮김, 김영사, 2017

**III. Discuss about process, thread, virtual memory and file**

* + 1. **Processes**

hello와 같은 프로그램이 실행될 때 OS는 Processor, Main Memory 및 I/O device 모두를 독점적으로 사용하는 것으로 보인다. Processor가 프로그램의 명령을 중단 없이 순서대로 실행하는 것처럼 나타난다. 그리고 프로그램의 코드와 데이터는 시스템의 memory에 있는 유일한 개체로 보일 수 있다. 이는 Computer Science에서 가장 중요한 아이디어 중 하나이다.

Process는 실행 중인 프로그램을 운영 체제가 추상화한 것이다. 여러 Process가 같은 시스템에서 동시에 실행될 수 있으며 각각은 hardware를 독점적으로 사용하는 것으로 나타난다. 한 process의 명령과 다른 process의 명령이 동시에 실행된다는 것을 의미한다.

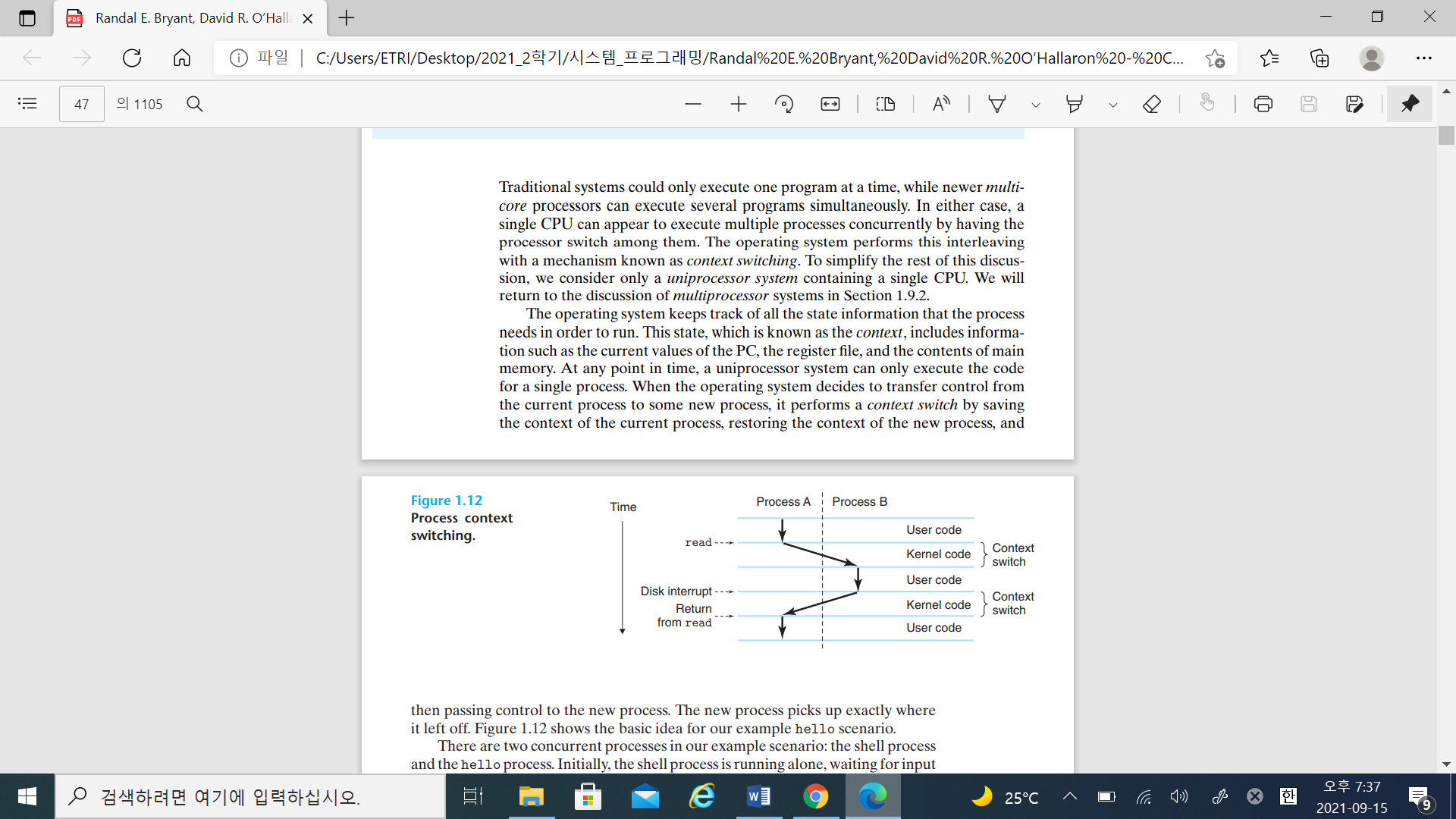


Figure 1.9 Process context switching

figure 1.9에서 알 수 있듯이, 한 process에서 다른 process로의 전환은 운영체제 *kernel*에 의해 관리된다. Kernel은 항상 메모리에 존재하는 운영 체제 코드의 일부이다. 응용 프로그램이 파일을 읽거나 쓰는 것과 같은 운영 체제에 의한 어떤 동작을 해야 할 때, kernel로 제어권을 전송하는 특별한 시스템 호출 명령을 실행한다. 이후 kernel은 요청된 작업을 수행하고 응용 프로그램으로 돌아간다. Kernel은 별도의 프로세스가 아니다. 대신 시스템이 모든 프로세스를 관리하기 위해 사용하는 코드 및 데이터 구조의 집합이다.

* + 1. **Threads**

일반적으로 process를 단일한 흐름으로 생각하지만, 현대 시스템에서 process는 실제로 process context에서 각각 실행되며 같은 코드와 글로벌 데이터를 공유하는 *threads* 라고 하는 여러 실행 단위로 구성된다. 네트워크 서버를 동시 실행할 것을 요구하기 때문에 threads는 점점 더 중요한 프로그래밍 모델이다. 여러 프로세스 간에 데이터를 공유하기 쉽고, 일반적으로 threads가 process보다 효율적이기 때문이다.

* + 1. **Virtual Memory**

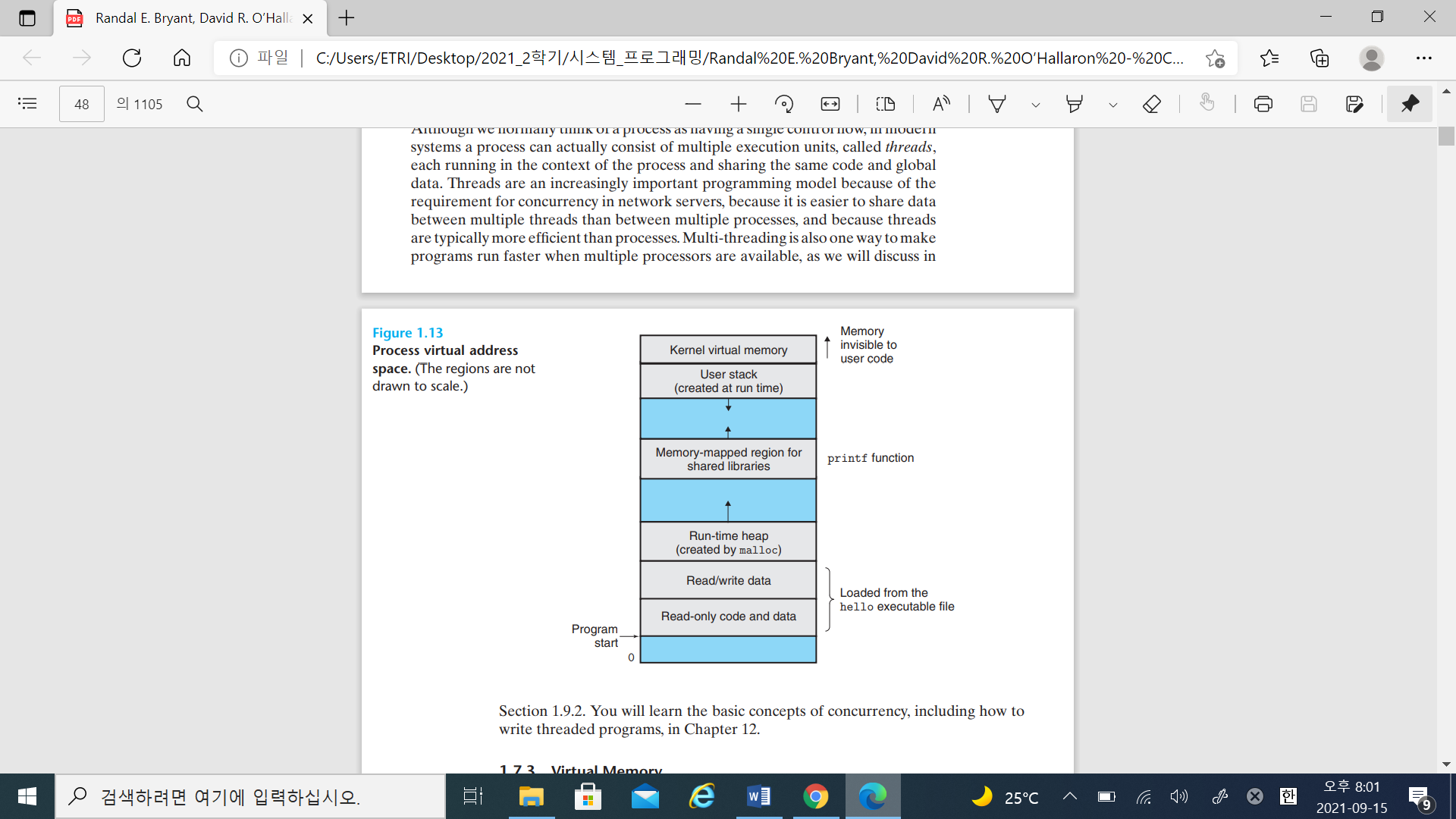


Figure 1.10 Virtual Address Space Process

Virtual memory는 각 process가 main memory를 독점적으로 사용하는 것처럼 보이게 하는 추상화이다. 각 process는 virtual address space로 알려진 같은 메모리 관점을 가진다. Linux process virtual address space는 그림 1.13에 나와 있다(다른 유닉스 시스템도 유사한 레이아웃을 사용한다). Linux에서 주소 공간의 맨 위 영역은 모든 process에 공통적인 운영 체제의 코드와 데이터를 위해 따로 잡아 둔다. 주소 공간의 하단 영역에는 사용자의 process에 의해 정의된 코드와 데이터가 있다.

각 process에서 볼 수 있는 Virtual Address Space는 다음과 같이 각각 특정한 목적을 가진 여러 개의 잘 정의된 영역으로 구성된다.

* Program code and data
* Heap
* Shared libraries
* Stack
* Kernel

**1.7.4 Files**

파일은 바이트의 나열이며 그 이상도 이하도 아니다. 디스크, 키보드, 디스플레이 및 네트워크를 포함한 모든 I/O 장치는 파일로 모델링 된다. 시스템의 모든 입력과 출력은 파일을 읽고 쓰면서 수행된다. 이 단순하고 우아한 파일에 대한 개념은 시스템에 포함될 수 있는 모든 다양한 I/O 장치를 application에서 균일하게 보기 때문에 매우 강력하다.