Operating-System Report

1팀
임과림, 정지오, 채지명, 박창선, 강미래

과목명.	운영체제론
담 당.	강경태 교수님
제출일.	2022년 5월 6일

한양대학교
HANYANG UNIVERSITY

Index

1. Introduction

1.1. What Operating Systems Do

1.1.1. User View

컴퓨터에 대한 사용자의 인식은 사용하는 인터페이스에 따라 다르다. 대부분의 컴퓨터 사용자는 모니터, 키보드, 마우스 및 시스템 장치로 구성된 PC 앞에 앉는다. 이 경우 운영 체제는 기본적으로 리소스 활용이 아닌 성능에만 중점을 두고 사용 편의성을 위해 설계되었다 — 다양한 하드웨어 및 소프트웨어 리소스를 공유하는 방법.

다른 경우에는 사용자가 메인프레임이나 미니컴퓨터에 연결된 터미널에 앉은다. 다른 사용자는 다른 터미널을 통해 동일한 컴퓨터에 액세스한다. 이러한 사용자는 리소스를 공유하고 정보를 교환할 수 있다. 이 경우 운영 체제는 리소스 활용을 극대화하도록 설계되었다 — 사용 가능한 모든 CPU 시간, 메모리 및 I/O가 효율적으로 사용되도록 리소스 활용도를 최대화하도록 설계되었다.

다음과 같은 다른 경우에 사용자는 다른 워크스테이션 및 서버의 네트워크에 연결된 워크스테이션에 앉아 있다. 이러한 사용자는 전용 리소스를 마음대로 사용할 수 있지만 파일, 컴퓨팅 및 인쇄 서버를 비롯한 네트워크 및 서버와 같은 리소스를 공유할 수 있다. 따라서 운영 체제는 개별 가용성과 리소스 활용도 사이에서 절충하도록 설계되었다.

최근에는 스마트폰, 태블릿PC 등 다양한 모바일 컴퓨터가 대중화되고 있다. 대부분의 모바일 컴퓨터는 개별 사용자를 위한 독립 실행형 장치이다. 많은 경우 셀룰러 또는 기타 무선 기술을 통해 네트워크에 연결한다. 이러한 모바일 장치는 주로 전자 메일 및 웹 검색을 위해 컴퓨터를 사용하는 데 관심이 있는 사람들을 위해 데스크톱 및 랩톱을 점차적으로 대체하고 있다. 모바일 컴퓨터의 사용자 인터페이스에는 일반적으로 사용자가 물리적 키보드와 마우스를 사용하는 대신 화면에서 손가락을 누르고 밀어 시스템과 상호 작용하는 터치 스크린이 있다.

일부 컴퓨터에는 사용자 보기가 거의 또는 전혀 없다. 예를 들어 가전 제품과 자동차에 내장된 컴퓨터에는 숫자 키패드가 있고 상태를 표시하기 위해 조명을 켜거나 끌 수 있지만 컴퓨터와 해당 운영 체제는 기본적으로 사용자 개입 없이 실행되도록 설계되었다.

1.1.2. System View

컴퓨터의 관점에서 운영 체제는 하드웨어와 가장 밀접한 프로그램이다. 이러한 맥락에서 운영 체제는 리소스 할당자로 생각할 수 있다. 컴퓨터 시스템에는 CPU 시간, 메모리 공간, 파일 저장 공간, I/O 장치 등 해결해야 할 수 있는 많은 리소스가 있다. 운영 체제는 이러한 리소스의 관리자 역할을 한다. 잠재적으로 충돌할 수 있는 수많은 리소스 요청에 직면하여 운영 체제는 컴퓨터 시스템을 효율적이고 공정하게 실행할 수 있도록 특정 프로그램과 사용자에게 리소스 요청을 할당하는 방법을 결정해야 한다. 리소스 할당은 많은 사용자가 동일한 메인프레임이나 미니컴퓨터에 액세스할 때 특히 중요하다.

운영 체제에 대한 약간 다른 관점은 다양한 I/O 장치와 사용자 프로그램을 제어할 필요성을 강조한다. 운영 체제는 제어 프로그램이다. 제어 프로그램은 컴퓨터의 오류 및 부적절한 사용을 방지하기 위해 사용자 프로그램의 실행을 관리한다. I/O 장치의 작동 및 제어에 특히 주의를 기울인다.

1.1.3. Defining Operating Systems

컴퓨터는 토스터기, 자동차, 선박, 우주선, 가정 및 기업에 존재한다. 그들은 게임 콘솔, 음악 플레이어, 케이블 TV 튜너 및 산업용 제어 시스템의 기초이다. 상대적으로 짧은 역사에도 불구하고 컴퓨터는 빠르게 발전했다. 컴퓨팅은 수행할 수 있는 작업을 결정하기 위한 실험으로 시작되었으며 암호 크래킹 및 궤적 매핑과 같은 군사용 고정 사용 시스템과 인구 조사와 같은 정부 사용으로 빠르게 이동했다. 이러한 초기 컴퓨터는 범용 다목적 메인프레임으로 진화했고, 이때 운영 체제가 탄생했다.

일반적으로 운영 체제는 완전히 적절하게 정의되지 않다. 운영 체제는 사용 가능한 컴퓨터 시스템을 만드는 문제를 해결하는 논리적인 방법을 제공하기 때문에 존재한다. 컴퓨터 시스템의 기본 목표는 사용자 프로그램을 실행하고 사용자 문제를 보다 쉽게 해결하는 것이다. 컴퓨터 하드웨어는 이 목표를 위해 만들어졌다. 베어 하드웨어만으로는 특별히 사용하기 쉽지 않기 때문에 응용 프로그램을 개발했다. 이러한 프로그램에는 I/O 장치의 작동 제어와 같은 특정 일반 작업이 필요하다. 그러면 리소스를 제어하고 할당하기 위한 공통 기능이 하나의 소프트웨어에 통합됨: 운영 체제.

또한 운영 체제 구성 요소에 대해 일반적으로 인정되는 정의가 없다. 간단한 요점은 "운영 체제"를 주문할 때 공급업체가 제공하는 모든 것이 포함되어 있다는 것이다. 그러나 포함된 기능은 시스템마다 크게 다른다. 일부 시스템은 메가바이트 미만을 차지하며 전체 화면 편집기도 없다. 다른 시스템은 기가바이트의 공간을 필요로 하며 전적으로 그래픽 창 시스템을 기반으로 한다. 보다 일반적인 정의와 일반적으로 따르는 것은 운영 체제에 의해 컴퓨터에서 항상 실행되는 프로그램 — 흔히 커널이라고 한다. (커널 외에도 두 가지 다른 유형의 프로그램이 있음: 운영 체제와 관련되지만 반드시 커널의 일부는 아닌 시스템 프로그램 및 시스템 작동과 관련되지 않은 모든 프로그램을 포함하는 응용 프로그램.)

오늘날 모바일 장치의 운영 체제를 살펴보면 작업을 구성하는 기능의 수가 증가하고 있음을 다시 한 번 알 수 있다. 모바일 운영 체제에는 일반적으로 핵심 커널뿐만 아니라 미들웨어도 포함 — 응용 프로그램 개발자에게 추가 서비스를 제공하는 소프트웨어 프레임워크 집합이다. 예를 들어, 가장 유명한 두 가지 모바일 운영 체제 — Apple의 IOS 및 Google의 Android — 이들 각각에는 데이터베이스, 멀티미디어, 그래픽 등을 지원하는 핵심 커널과 미들웨어가 있다.

1.2. Computer-System Organization

1.2.1. Computer-System Operation

현대의 범용 컴퓨터 시스템은 하나 이상의 CPU와 여러 장치 컨트롤러로 구성된다. 이러한 장치 컨트롤러는 공유 메모리에 대한 액세스를 제공하는 공통 버스로 연결된다. 각 장치 컨트롤러는 특정 유형의 장치를 담당한다.(예: 디스크 드라이브, 오디오 장비 또는 비디오 모니터). CPU와 장치 컨트롤러는 메모리 주기를 놓고 경쟁하면서 병렬로 실행할 수 있다. 공유 메모리에 대한 규칙적인 액세스를 보장하기 위해 메모리 컨트롤러는 메모리에 대한 액세스를 동기화한다.

컴퓨터가 실행을 시작하려고 한다 — 예를 들어 전원을 켜거나 다시 시작할 때 실행할 초기 프로그램이 있어야 한다. 이 초기 프로그램이나 부트스트랩은 종종 간단한다. 일반적으로 컴퓨터 하드웨어의 읽기 전용 메모리(ROM) 또는 전기적으로 지울 수 있는 프로그래밍 가능한 읽기 전용 메모리(EEPROM)에 저장된다. CPU 레지스터에서 장치 컨트롤러, 메모리 내용에 이르기까지 시스템의 모든 측면을 초기화한다. 부트로더는 운영 체제를 로드하는 방법과 해당 시스템 실행을 시작하는 방법을 알아야 한다. 이를 달성하기 위해 부트로더는 운영 체제 커널을 찾아 메모리에 로드해야 한다.

커널이 로드되고 실행되면 시스템과 사용자에게 서비스를 제공할 수 있다. 일부 서비스는 부팅 시 메모리에 로드되어 시스템 프로세스가 되는 시스템 프로그램이나 커널이 실행되는 전체 시간 동안 실행되는 시스템 데몬에 의해 커널 외부에서 제공된다. UNIX에서 첫 번째 시스템 프로세스는 다른 많은 데몬을 시작하는 "init"이다. 이 단계가 완료되면 시스템이 완전히 부팅되고 이벤트가 발생할 때까지 기다린다.

이벤트의 발생은 일반적으로 하드웨어 또는 소프트웨어 인터럽트에 의해 신호를 받습니다. 하드웨어는 일반적으로 시스템 버스를 통해 CPU에 신호를 보내 언제든지 인터럽트를 트리거할 수 있습니다. 소프트웨어는 시스템 호출이라는(모니터 호출이라고도 함) 특수 작업을 수행하여 인터럽트를 트리거할 수 있습니다.

CPU가 인터럽트되면 수행하던 작업을 중지하고 즉시 실행을 고정된 위치로 옮긴다. 고정 위치는 일반적으로 인터럽트 서비스 루틴이 있는 시작 주소를 포함한다. 인터럽트 서비스 루틴 실행; 완료되면 CPU는 중단된 계산을 재개한다.

인터럽트는 컴퓨터 시스템 아키텍처의 중요한 부분이다. 모든 컴퓨터 설계에는 고유한 인터럽트 메커니즘이 있지만 몇 가지 기능이 공통적이다. 인터럽트는 적절한 인터럽트 서비스 루틴으로 제어를 전달해야 한다. 이 전송을 처리하는 간단한 방법은 일반 루틴을 호출하여 인터럽트 정보를 확인하는 것이다. 인터럽트 아키텍처는 인터럽트된 명령어의 주소도 저장해야 한다. 최신 아키텍처는 시스템 스택에 반환 주소를 저장한다. 인터럽트 루틴이 프로세서 상태를 수정해야 하는 경우 — 예를 들어 레지스터 값을 수정한다 — 현재 상태를 명시적으로 저장한 다음 반환하기 전에 해당 상태를 복원해야 한다. 인터럽트가 서비스된 후 저장된 리턴 주소가 프로그램 카운터에 로드되고 인터럽트 계산은 인터럽트가 발생하지 않은 것처럼 계속된다.

1.2.2. Storage Structure

CPU는 메모리에서 명령어만 로드할 수 있으므로 실행할 모든 프로그램을 메모리에 저장해야 합니다. 범용 컴퓨터는 주 메모리라고 하는 다시 쓰기 가능한 메모리에서 대부분의 프로그램을 실행한다. 주 메모리는 일반적으로 동적 랜덤 액세스 메모리라는(DRAM) 반도체 기술로 구현된다.

컴퓨터는 다른 형태의 메모리도 사용한다. ROM은 변경할 수 없으므로 앞에서 설명한 부트로더와 같은 정적 프로그램만 저장된다. EEPROM은 대체될 수 있지만, 주로 정적 프로그램을 포함하기 때문에 자주는 아니다. 예를 들어, 스마트폰에는 공장에서 설치된 프로그램을 저장하기 위한 EEPROM이 있다.

모든 형태의 메모리는 바이트 배열을 제공한다. 각 바이트에는 고유한 주소가 있다. 상호 작용은 특정 메모리 주소에 대한 일련의 로드 또는 저장 명령을 통해 이루어진다. 로드 명령어는 바이트 또는 워드를 주 메모리에서 CPU의 내부 레지스터로 이동하는 반면 저장 명령어는 레지스터의 내용을 주 메모리로 이동한다. 명시적 로드 및 저장 외에도 CPU는 공동 실행을 위해 주 메모리에서 명령을 자동으로 로드한다.

프로그램이 메모리 주소를 생성하는 방법은 무시할 수 있다. 실행 중인 프로그램에 의해 생성된 메모리 주소 시퀀스에만 관심이 있다. 이상적으로는 프로그램과 데이터가 주 메모리에 영구적으로 상주하기를 원한다. 이 배열은 일반적으로 두 가지 이유로 불가능하다.
주기억장치는 필요한 모든 프로그램과 데이터를 저장하기에는 너무 작은 경우가 많다.
주기억장치는 전원을 끄거나 기타 다른 방법으로 손실되는 휘발성 저장 장치이다.
따라서 대부분의 컴퓨터 시스템은 주 기억 장치의 확장으로 보조 기억 장치를 제공한다. 보조 저장소의 기본 요구 사항은 많은 양의 데이터를 영구적으로 보관할 수 있는 능력이다.

속도와 비용의 차이 외에도 다양한 스토리지 시스템은 휘발성이거나 비휘발성이다. 장치의 전원이 꺼지면 휘발성 저장소의 내용이 손실된다. 고가의 배터리 및 발전기 백업 시스템이 없는 경우 데이터를 비휘발성 메모리에 기록하여 안전하게 보관해야 한다.
완전한 메모리 시스템 설계는 가능한 한 많은 값비싼 비휘발성 메모리를 제공하면서 가능한 한 많은 값비싼 메모리를 사용해야 한다. 두 구성 요소 간의 액세스 시간이나 전송 속도에 큰 차이가 있는 경우 성능을 향상시키기 위해 캐싱을 설치할 수 있다.

1.2.3. I/O Structure

스토리지는 컴퓨터에 있는 여러 유형의 I/O 장치 중 하나일 뿐이다. 운영 체제 코드의 상당 부분은 시스템 안정성과 성능에 대한 중요성과 장치의 특성이 다르기 때문에 I/O 관리에 사용된다.

범용 컴퓨터 시스템은 공통 버스로 연결된 CPU와 여러 장치 컨트롤러로 구성된다. 각 장치 컨트롤러는 특정 유형의 장치를 담당한다. 컨트롤러에 따라 여러 장치가 연결될 수 있다. 예를 들어, 7개 이상의 장치를 소형 컴퓨터 시스템 인터페이스(SCSI) 컨트롤러에 연결할 수 있다. 장치 컨트롤러는 일부 로컬 버퍼 저장소와 특수 레지스터 집합을 유지 관리한다. 장치

컨트롤러는 제어하는 주변 장치와 로컬 버퍼 메모리 간에 데이터를 이동하는 역할을 한다. 일반적으로 운영 체제에는 각 장치 컨트롤러에 대해 하나의 장치 드라이버가 있다. 이 장치 드라이버는 장치 컨트롤러를 이해하고 나머지 운영 체제에 장치에 대한 통합 인터페이스를 제공한다.

I/O 작업을 시작하기 위해 장치 드라이버는 장치 컨트롤러에서 적절한 레지스터를 로드한다. 장치 컨트롤러는 차례로 이러한 레지스터의 내용을 확인하여 수행할 작업(예: "키보드에서 문자 읽기")을 결정한다. 컨트롤러는 장치에서 로컬 버퍼로 데이터 전송을 시작한다. 데이터 전송이 완료되면 장치 컨트롤러는 인터럽트를 통해 장치 드라이버에게 작업이 완료되었음을 알린다. 그런 다음 장치 드라이버는 운영 체제에 제어를 반환한다. 작업이 읽기인 경우 데이터 또는 데이터에 대한 포인터가 될 수 있다. 다른 작업의 경우 장치 드라이버가 상태 정보를 반환한다.

이러한 형태의 인터럽트 구동 I/O는 소량의 데이터를 이동하는 데 적합하지만 디스크 I/O와 같은 대량 데이터 이동에 사용할 경우 높은 오버헤드가 발생한다. 이 문제를 해결하기 위해 직접 메모리 액세스가 사용된다(DMA). I/O 장치에 대한 버퍼, 포인터 및 카운터를 설정한 후 장치 컨트롤러는 CPU 개입 없이 전체 데이터 블록을 자체 버퍼 메모리로 또는 자체 버퍼 메모리에서 직접 전송한다. 저속 장치에 대해 바이트당 하나의 인터럽트를 생성하는 대신 블록당 하나의 인터럽트만 생성하여 장치 드라이버에 작업이 완료되었음을 알린다. 장치 컨트롤러가 이러한 작업을 수행하는 동안 CPU는 다른 작업에 사용될 수 있다.
일부 고급 시스템은 버스 아키텍처가 아닌 스위치를 사용한다. 이러한 시스템에서 여러 구성 요소는 공유 버스에서 주기를 놓고 경쟁하지 않고 동시에 다른 구성 요소와 통신할 수 있다. 이 경우 DMA가 더 효율적이다.

1.3. Computer-System Architecture

1.3.1. Single-Processor Systems

대부분의 컴퓨터 시스템은 단일 프로세서를 사용한다. 단일 프로세서 시스템에는 사용자 프로세스의 명령을 포함하여 범용 명령 집합을 실행할 수 있는 주 CPU가 있다. 거의 모든 단일 프로세서 시스템에는 다른 특수 프로세서도 있다. 장치별 프로세서의 형태로 제공될 수 있다. 예를 들어 디스크, 키보드 및 그래픽 컨트롤러가 있다. 또는 메인프레임에서 시스템 구성 요소 간에 데이터를 빠르게 이동하는 I/O 프로세서와 같은 보다 범용 프로세서의 형태로 제공될 수 있다.

이러한 모든 특수 프로세서는 제한된 명령 집합을 실행하고 사용자 프로세스를 실행하지 않다. 운영 체제가 다음 작업에 대한 정보를 보내고 상태를 모니터링하기 때문에 운영 체제에서 관리하는 경우가 있다. 예를 들어, 디스크 컨트롤러 마이크로프로세서는 주 CPU로부터 일련의 요청을 수신하고 자체 디스크 대기열 및 스케줄링 알고리즘을 구현한다. 이 배열은 주 CPU의 디스크 스케줄링 오버헤드를 완화한다. PC의 키보드에는 키 입력을 CPU로 보낼 코드로 변환하는 마이크로프로세서가 있다. 다른 시스템이나 환경에서 특수 목적 프로세서는

하드웨어에 내장된 저수준 구성 요소이다. 운영 체제는 이러한 프로세서와 통신할 수 없다. 그들은 자율적으로 일을 한다. 전용 마이크로프로세서의 사용은 일반적이며 단일 프로세서 시스템을 다중 프로세서로 바꾸지 않다. 범용 CPU가 하나만 있는 경우 시스템은 단일 프로세서 시스템이다.

1.3.2. Multiprocessor Systems

지난 몇 년 동안 멀티프로세서 시스템(병렬 시스템 또는 멀티코어 시스템이라고도 함)이 컴퓨팅 환경을 지배하게 되었다. 이러한 시스템에는 컴퓨터 버스와 때로는 시계, 메모리 및 주변 장치를 공유하는 긴밀한 통신에 두 개 이상의 프로세서가 있다. 멀티프로세서 시스템은 서버에 처음 등장한 후 데스크탑 및 랩탑 시스템으로 마이그레이션되었다. 최근에는 스마트폰 및 태블릿과 같은 모바일 장치에 멀티프로세서가 등장했다. 다중 프로세서 시스템에는 세 가지 주요 이점이 있다.

처리량을 늘린다. 프로세서 수를 늘리면 더 짧은 시간에 더 많은 작업을 수행할 수 있다. 그러나 N 프로세서의 속도 향상은 N이 아니라 N보다 작은다. 여러 프로세서가 작업에 협력할 때 모든 부품이 계속 작동하도록 하는 오버헤드가 있다. 공유 리소스에 대한 경합과 함께 이 오버헤드는 추가 프로세서의 예상 이점을 줄이다. 유사하게, 긴밀하게 협력하는 N명의 프로그래머는 단일 프로그래머가 하는 것보다 N배 많은 작업을 생성하지 않다.

규모의 경제. 다중 프로세서 시스템은 주변 장치, 대용량 저장 장치 및 전원을 공유할 수 있기 때문에 동등한 다중 단일 프로세서 시스템보다 비용이 저렴할 수 있다. 여러 프로그램이 동일한 데이터 세트에서 작업하는 경우 많은 로컬 컴퓨터 디스크와 데이터 복사본을 많이 사용하는 대신 해당 데이터를 하나의 디스크에 저장하고 모든 프로세서가 이를 공유하도록 한다.

신뢰성을 향상시킨다. 기능이 여러 프로세서에 적절하게 분산될 수 있다면 하나의 프로세서에 장애가 발생해도 시스템이 중지되지 않고 속도가 느려질 뿐이다. 10개의 프로세서가 있고 하나가 실패하면 나머지 9개의 프로세서는 각각 실패한 프로세서를 공유할 수 있다. 결과적으로 전체 시스템은 완전한 오류가 아니라 10%만 느리게 실행된다.

많은 응용 프로그램에서 컴퓨터 시스템의 안정성을 개선하는 것이 중요한다. 살아남은 하드웨어 수준에 비례하여 서비스를 계속 제공할 수 있는 능력을 단계적 성능 저하라고 한다. 일부 시스템은 정상적인 성능 저하를 넘어 단일 구성 요소에 장애가 발생해도 계속 작동할 수 있기 때문에 내결함성이라고 한다. 내결함성은 오류를 감지, 진단 및 수정하는 메커니즘이 필요한다. HP NonStop(이전 Tandem) 시스템은 하드웨어 및 소프트웨어 복제를 모두 사용하여 장애가 발생한 경우에도 지속적인 작동을 보장한다. 시스템은 동기식으로 작동하는 여러 쌍의 CPU로 구성된다. 쌍의 두 프로세서는 각 명령을 실행하고 결과를 비교한다. 결과가 다른 경우 쌍의 CPU 중 하나에 장애가 발생하고 둘 다 중지된다. 그런 다음 실행 프로세스가 다른 CPU 쌍으로 이동되고 실패한 명령이 다시 시작된다. 이 솔루션은 특수 하드웨어와 많은 하드웨어 중복을 포함하기 때문에 비용이 많이든다.

대칭 멀티프로세싱과 비대칭 멀티프로세싱의 차이점은 하드웨어나 소프트웨어에서 비롯된다. 특수 하드웨어는 여러 프로세서를 구별하거나 한 명의 보스와 여러 작업자만 허용하도록 소프트웨어를 작성할 수 있다.

CPU 설계의 최신 트렌드는 단일 칩에 여러 컴퓨팅 코어를 포함하는 것이다. 이러한 다중 프로세서 시스템을 다중 코어라고 한다. 온칩 통신이 칩 간 통신보다 빠르기 때문에 단일 코어가 있는 여러 칩보다 효율적이다. 또한 다중 코어 칩의 전력 소비는 다중 단일 코어 칩의 전력 소비보다 훨씬 낮은다. 멀티 코어 시스템은 멀티 프로세서 시스템이지만 모든 멀티 프로세서 시스템이 멀티 코어인 것은 아니다.

마지막으로 블레이드 서버는 여러 프로세서 보드, I/O 보드 및 네트워크 보드가 동일한 섀시에 배치되는 비교적 최근에 개발된 것이다. 이러한 다중 프로세서 시스템과 기존 다중 프로세서 시스템 간의 차이점은 각 블레이드 프로세서 보드가 독립적으로 부팅되고 자체 운영 체제를 실행한다는 것이다. 일부 블레이드 서버 마더보드는 컴퓨터 유형 간의 경계를 모호하게 하는 다중 프로세서이기도 한다. 기본적으로 이러한 서버는 여러 개의 독립적인 다중 프로세서 시스템으로 구성된다.

1.3.3. Clustered Systems

다중 프로세서 시스템의 또 다른 유형은 여러 CPU를 함께 집계하는 클러스터 시스템이다. 클러스터 시스템은 함께 연결된 둘 이상의 개별 시스템(또는 노드)으로 구성된다는 점에서 섹션 1.3.2에 설명된 다중 프로세서 시스템과 다르다. 이러한 시스템은 느슨하게 결합된 것으로 간주된다. 각 노드는 단일 프로세서 시스템 또는 다중 코어 시스템일 수 있다. 클러스터의 정의가 구체적이지 않다는 점에 유의하는 것이 중요한다. 많은 상용 소프트웨어 패키지는 클러스터 시스템을 정의하는 데 어려움을 겪고 있으며 한 형식이 다른 형식보다 나은 이유를 설명한다. 일반적으로 인정되는 정의는 저장소를 공유하고 LAN 또는 더 빠른 상호 연결(InfiniBand)로 밀접하게 연결된 컴퓨터 클러스터이다.

클러스터는 고가용성 서비스를 제공하는 데 자주 사용된다. 즉, 클러스터에 있는 하나 이상의 시스템에 장애가 발생하더라도 서비스는 계속된다. 일반적으로 시스템에 어느 정도 중복성을 추가하여 고가용성을 달성한다. 클러스터 소프트웨어 계층은 클러스터 노드에서 실행된다. 각 노드는 하나 이상의 다른 노드를 모니터링할 수 있다(LAN을 통함). 모니터링되는 시스템이 실패하면 모니터링 시스템이 스토리지의 소유권을 가져오고 실패한 시스템에서 실행 중인 애플리케이션을 다시 시작할 수 있다. 응용 프로그램의 사용자와 클라이언트는 서비스가 잠시 중단되는 것을 볼 수 있다.

클러스터는 비대칭 또는 대칭으로 구성할 수 있다. 비대칭 클러스터에서 한 시스템은 상시 대기 모드에 있고 다른 시스템은 애플리케이션을 실행하고 있다. 상시 대기 호스트는 활성 서버를 모니터링하는 것 외에는 아무 작업도 수행하지 않다. 해당 서버에 장애가 발생하면 상시 대기 호스트가 활성 서버가 된다. 대칭 클러스터에서 둘 이상의 호스트가 애플리케이션을 실행하고 서로를 모니터링한다. 이 구조는 사용 가능한 모든 하드웨어를 사용하기 때문에 분명히 더 효율적이다. 그러나 실행하려면 둘 이상의 응용 프로그램이 필요하다.

클러스터는 네트워크로 연결된 여러 컴퓨터 시스템으로 구성되어 있기 때문에 고성능 컴퓨팅 환경을 제공하는 데 사용할 수도 있다. 이러한 시스템은 클러스터의 모든 컴퓨터에서 동시에 응용 프로그램을 실행할 수 있기 때문에 단일 프로세서 또는 SMP 시스템보다 더 많은 컴퓨팅 성능을 제공할 수 있다. 그러나 클러스터를 활용하려면 애플리케이션을 특별히 작성해야 합니다.

여기에는 클러스터의 개별 컴퓨터에서 병렬로 실행되는 개별 구성 요소로 프로그램을 나누는 병렬화라는 기술이 포함된다. 일반적으로 이러한 애플리케이션은 클러스터의 각 컴퓨팅 노드가 문제의 일부를 해결하면 모든 노드의 결과가 최종 솔루션으로 결합되도록 설계되었다.

클러스터링의 다른 형태에는 광역 네트워크를 통한 병렬 클러스터 및 클러스터(WAN)가 포함된다. 병렬 클러스터를 사용하면 여러 호스트가 공유 스토리지의 동일한 데이터에 액세스할 수 있다. 대부분의 운영 체제는 동시에 데이터에 액세스하는 여러 호스트에 대한 지원이 부족하기 때문에 병렬 클러스터는 종종 특수 버전의 소프트웨어 및 응용 프로그램을 사용해야 한다. 예를 들어, Oracle Real Application Cluster는 병렬 클러스터에서 실행되도록 설계된 Oracle Database 버전이다. 각 시스템은 Oracle을 실행하며 공유 디스크에 대한 액세스를 추적하는 소프트웨어 계층이 있다. 각 시스템은 데이터베이스의 모든 데이터에 대한 전체 액세스 권한을 갖는다.

1.4. Operating-System Structure

운영 체제의 가장 중요한 측면 중 하나는 다중 프로그래밍 기능이다. 일반적으로 단일 프로그램은 CPU 또는 I/O 장치를 항상 사용 중인 상태로 유지할 수 없다. 한 명의 사용자가 여러 프로그램을 실행하는 경우가 많다. 멀티프로그래밍은 CPU가 항상 실행할 작업을 갖도록 작업(코드 및 데이터)을 구성하여 CPU 사용률을 향상시킨다.

메모리의 작업 집합은 작업 풀에 있는 작업의 하위 집합일 수 있다. 운영 체제는 메모리에서 작업을 선택하고 실행을 시작한다. 결국 작업은 특정 작업(예: I/O 작업)이 완료될 때까지 기다려야 할 수 있다. 다중 프로그래밍되지 않은 시스템에서 CPU는 유휴 상태가 된다. 다중 프로그램 시스템에서 운영 체제는 다른 작업으로 전환하여 실행한다. 해당 작업이 기다려야 할 때 CPU는 다른 작업으로 전환하는 식이다. 결국 첫 번째 작업은 대기를 끝내고 CPU를 다시 가져온다. 실행할 작업이 하나 이상 있는 한 CPU는 유휴 상태가 아니다.
멀티프로그래밍 시스템은 다양한 시스템 자원(예: CPU, 메모리, 주변기기)이 효율적으로 활용되는 환경을 제공하지만 컴퓨터 시스템과의 사용자 상호작용은 제공하지 않다. 시분할(또는 멀티태스킹)은 멀티프로그래밍의 논리적 확장이다. 시분할 시스템에서 CPU는 여러 작업을 전환하여 여러 작업을 실행하지만 전환이 너무 자주 발생하여 사용자가 실행될 때 각 프로그램과 상호 작용할 수 있다.

시간 공유를 위해서는 사용자와 시스템 간의 직접적인 통신을 제공하는 대화형 컴퓨터 시스템이 필요하다. 사용자는 키보드, 마우스, 터치패드 또는 터치스크린과 같은 입력 장치를 사용하여 운영 체제나 프로그램에 직접 지시를 내리고 출력 장치에서 즉각적인 결과를 기다린다. 따라서 응답 시간은 일반적으로 1초 미만으로 짧아야 한다.

시분할 운영 체제를 사용하면 많은 사용자가 동시에 컴퓨터를 공유할 수 있다. 시분할 시스템의 각 동작이나 명령은 짧은 경향이 있기 때문에 사용자당 필요한 CPU 시간은 매우 적은다.

시스템이 한 사용자에서 다음 사용자로 빠르게 전환되면 각 사용자는 전체 컴퓨터 시스템이 여러 사용자 간에 공유되더라도 전체 컴퓨터 시스템이 자신의 용도에 전념한다는 인상을 줄 것이다.

시분할 운영 체제는 CPU 스케줄링 및 다중 프로그래밍을 사용하여 각 사용자에게 시분할 컴퓨터의 일부를 제공한다. 각 사용자는 메모리에 최소한 하나의 개별 프로그램을 가지고 있다. 메모리에 로드되어 실행되는 프로그램을 프로세스라고 한다. 프로세스가 실행될 때 일반적으로 완료되거나 I/O를 수행해야 하기 전에 짧은 시간 동안만 실행된다. I/O는 대화식일 수 있다. 즉, 출력은 사용자의 디스플레이로 이동하고 입력은 사용자의 키보드, 마우스 또는 기타 장치에서 나온다. 대화형 I/O는 종종 "인간의 속도로" 실행되기 때문에 완료하는 데 오랜 시간이 걸릴 수 있다.

1.5. Operating-System Operations

1.5.1. Dual-Mode and Multimode Operation

최소한 사용자 모드와 커널 모드(감독자 모드, 시스템 모드 또는 특권 모드라고도 함)의 두 가지 별도의 작동 모드가 필요하다. 현재 모드인 커널(0) 또는 사용자(1)를 나타내기 위해 모드 비트라고 하는 비트가 컴퓨터의 하드웨어에 추가된다. 모드 비트를 사용하여 운영 체제를 대신하여 수행되는 작업과 사용자를 대신하여 수행되는 작업을 구분할 수 있다. 컴퓨터 시스템이 사용자 응용 프로그램을 대신하여 실행될 때 시스템은 사용자 모드에 있다. 그러나 사용자 응용 프로그램이 시스템 호출을 통해 운영 체제에서 서비스를 요청할 때 시스템은 요청을 완료하기 위해 사용자 모드에서 커널 모드로 전환해야 한다.

시스템 시작 시 하드웨어는 커널 모드에서 시작된다. 그런 다음 운영 체제를 로드하고 사용자 모드에서 사용자 응용 프로그램을 시작한다. 트랩이나 인터럽트가 발생할 때마다 하드웨어는 사용자 모드에서 커널 모드로 전환한다(즉, 모드 비트의 상태를 0으로 변경). 따라서 운영 체제가 컴퓨터를 제어하는 한 커널 모드에 있게 된다. 시스템은 사용자 프로그램에 제어를 넘기기 전에 항상 사용자 모드로 전환한다(모드 비트를 1로 설정).

1.5.2. Timer

사용자에게 제어권을 넘기기 전에 OS는 타이머가 인터럽트로 설정되어 있는지 확인한다. 타이머가 중단되면 제어가 OS에 자동으로 넘어가므로 인터럽트를 치명적인 오류로 간주하거나 프로그램에 더 많은 시간을 줄 수 있다. 분명히 타이머의 내용을 수정하는 명령은 권한이 있다.

사용자 프로그램이 너무 오래 실행되는 것을 방지하기 위해 타이머를 사용할 수 있다. 간단한 기술은 프로그램 실행이 허용된 시간으로 카운터를 초기화하는 것이다. 예를 들어 시간 제한이 7분인 프로그램은 카운터를 420으로 초기화한다. 1초마다 타이머가 중단되고 카운터가 1씩

감소한다. 카운터가 양수인 한 제어는 사용자 프로그램으로 돌아간다. 카운터가 음수가 되면 운영 체제는 지정된 시간 제한을 초과하여 프로그램을 종료한다.

1.6. Process Management

프로그램은 명령이 CPU에 의해 실행되지 않는 한 아무 일도 하지 않다. 앞서 언급했듯이 실행 프로그램은 프로세스이다. 컴파일러와 같은 시분할 사용자 프로그램은 프로세스이다. 개인 사용자가 PC에서 실행하는 워드 프로세싱 프로그램은 프로세스이다. 출력을 프린터로 보내는 것과 같은 시스템 작업은 프로세스(또는 적어도 프로세스의 일부)일 수도 있다.

프로세스가 작업을 완료하려면 CPU 시간, 메모리, 파일 및 I/O 장치를 비롯한 특정 리소스가 필요하다. 이러한 리소스는 생성 시 프로세스에 할당되거나 런타임에 할당된다. 프로세스가 생성될 때 얻는 다양한 물리적 및 논리적 리소스 외에도 다양한 초기화 데이터(입력)도 전달할 수 있다. 예를 들어, 터미널 화면에 파일의 상태를 표시하는 기능을 가진 프로세스를 고려한다. 프로세스는 파일 이름을 입력으로 받고 적절한 명령과 시스템 호출을 실행하여 터미널에 필요한 정보를 가져와 표시한다. 프로세스가 종료되면 운영 체제는 재사용 가능한 모든 리소스를 회수한다.

프로세스는 시스템의 작업 단위이다. 시스템은 일련의 프로세스로 구성되며 그 중 일부는 운영 체제 프로세스(시스템 코드를 실행하는 프로세스)이고 나머지는 사용자 프로세스(사용자 코드를 실행하는 프로세스)이다. 이러한 모든 프로세스는 예를 들어 단일 CPU에서 다중화하여 동시에 실행할 수 있다.

운영 체제는 프로세스 관리와 관련된 다음 활동을 담당한다.

- CPU에서 프로세스 및 스레드를 예약한다.
- 사용자 및 시스템 프로세스를 만들고 삭제한다.
- 프로세스를 일시 중지하고 다시 시작한다.
- 프로세스 동기화 메커니즘을 제공한다.
- 프로세스 통신 메커니즘을 제공한다.

1.7. Memory Management

CPU 활용도와 사용자에 대한 컴퓨터의 응답성을 향상시키기 위해 범용 컴퓨터는 메모리에 여러 프로그램을 유지해야 하며, 이는 메모리 관리가 필요하다. 다양한 메모리 관리 체계가 사용된다. 이러한 체계는 다양한 접근 방식을 반영하며 주어진 알고리즘의 효율성은 특정 상황에 따라

다른다. 특정 시스템에 대한 메모리 관리 체계를 선택할 때 많은 요소, 특히 시스템의 하드웨어 설계를 고려해야 한다. 각 알고리즘에는 자체 하드웨어 지원이 필요하다.

운영 체제는 메모리 관리와 관련된 다음 활동을 담당한다.

- 메모리의 현재 사용 중인 부분과 사용 중인 부분을 추적한다.
- 메모리 안팎으로 이동할 프로세스(또는 프로세스의 일부)와 데이터를 결정한다.
- 필요에 따라 메모리 공간을 할당하고 해제한다.

1.8. Storage Management

1.8.1. File-System Management

파일 관리는 운영 체제에서 가장 눈에 띄는 구성 요소 중 하나이다. 컴퓨터는 여러 유형의 물리적 매체에 정보를 저장할 수 있다. 디스크, CD 및 테이프가 가장 일반적이다. 이러한 각 미디어에는 고유한 특성과 물리적 구성이 있다. 각 매체는 고유한 특성을 가진 디스크 드라이브나 테이프 드라이브와 같은 장치에 의해 제어된다. 이러한 속성에는 액세스 속도, 용량, 데이터 전송 속도 및 액세스 방법(순차 또는 임의)이 포함된다.

운영 체제는 파일 관리와 관련된 다음 활동을 담당한다.

- 파일을 만들고 삭제한다.
- 파일을 구성할 디렉토리를 만들고 삭제한다.
- 파일 및 디렉토리 조작을 위한 기본 요소를 지원한다.
- 파일을 보조 저장소에 매핑한다.
- 안정적인(비휘발성) 저장 매체에 파일을 백업한다.

1.8.2. Mass-Storage Management

주 메모리는 모든 데이터와 프로그램을 담기에는 너무 작고, 정전이 발생하면 보유하고 있는 데이터가 손실되기 때문에 컴퓨터 시스템은 주 메모리를 백업하기 위해 보조 저장 장치를 제공해야 한다. 대부분의 최신 컴퓨터 시스템은 디스크를 프로그램 및 데이터의 기본 온라인 저장 매체로 사용한다. 컴파일러, 어셈블러, 워드 프로세서, 편집기 및 포맷터를 포함한 대부분의 프로그램은 메모리에 로드될 때까지 디스크에 저장된다. 그런 다음 처리를 위한 소스 및 대상으로 디스크를 사용한다. 따라서 디스크 스토리지의 적절한 관리는 컴퓨터 시스템에 매우 중요한다. 스토리지는 컴퓨터 시스템에 매우 중요한다. 운영 체제는 디스크 관리와 관련된 다음 활동을 담당한다.

- 여유 공간 관리
- 스토리지 할당
- 디스크 스케줄링

보조기억장치는 자주 사용되기 때문에 효율적으로 사용해야 한다. 컴퓨터의 전체 작동 속도는 디스크 하위 시스템의 속도와 해당 하위 시스템을 조작하는 알고리즘에 따라 달라질 수 있다.

그러나 느리고 저렴한 스토리지는 2차 스토리지에 비해 용도가 많다. 디스크 데이터의 백업, 거의 사용하지 않는 데이터의 저장, 장기 보관 저장이 몇 가지 예이다. 테이프 드라이브와 테이프, CD 및 DVD 드라이브와 플래터는 일반적인 3차 저장 장치이다. 미디어(테이프 및 디스크)는 WORM(한 번 쓰기, 여러 번 읽기) 형식과 RW(읽기-쓰기) 형식이 다르다.

3차 스토리지는 시스템 성능에 중요하지 않지만 여전히 관리되어야 한다. 일부 운영 체제는 이 작업을 수행하는 반면 다른 운영 체제는 3차 스토리지 관리를 애플리케이션에 맡긴다. 운영 체제가 제공할 수 있는 일부 기능에는 장치에 미디어 마운트 및 마운트 해제, 프로세스에서 독점적으로 사용할 장치 할당 및 해제, 2차 저장소에서 3차 저장소로 데이터 마이그레이션 등이 있다.

1.8.3. Caching

캐싱은 컴퓨터 시스템의 중요한 원리이다. 작동 방식은 다음과 같다. 정보는 일반적으로 일부 스토리지 시스템에 보관된다. 사용 중에는 더 빠른 스토리지 시스템에 임시로 복사된다 — 캐싱 중. 특정 정보가 필요할 때 먼저 캐시에 있는지 확인한다. 그렇다면 캐시의 정보를 직접 사용한다. 그렇지 않은 경우 소스의 정보를 사용하여 곧 다시 필요할 것이라는 가정 하에 복사본을 캐시에 넣다.

1.8.4. I/O Systems

운영 체제의 목적 중 하나는 특정 하드웨어 장치의 특성을 사용자에게 숨기는 것이다. 예를 들어, UNIX에서 I/O 장치의 특성은 I/O 하위 시스템에 의해 대부분의 운영 체제 자체에서 숨겨진다. I/O 하위 시스템은 다음과 같은 여러 구성 요소로 구성된다.

- 버퍼링, 캐싱 및 스풀링을 포함하는 메모리 관리 구성 요소
- 일반 장치 드라이버 인터페이스
- 특정 하드웨어 장치용 드라이버
- 장치 드라이버만 할당된 특정 장치의 특성을 알고 있다.

1.9. Protection and Security

보호는 컴퓨터 시스템에서 정의한 리소스에 대한 프로세스 또는 사용자의 액세스를 제어하는 모든 메커니즘이다. 메커니즘은 부과할 제어를 지정하고 이러한 제어를 시행하는 방법을 제공해야 한다.

보호는 구성 요소 하위 시스템 간의 인터페이스에서 잠재적인 오류를 감지하여 안정성을 향상시킬 수 있다. 인터페이스 오류를 조기에 감지하면 종종 정상적인 하위 시스템이 다른 결함이 있는 하위 시스템에 의해 오염되는 것을 방지할 수 있다. 또한 보호되지 않은 리소스는 권한이 없거나 무능한 사용자의 사용(또는 오용)을 방지하지 않다.

시스템은 적절하게 보호될 수 있지만 여전히 실패하기 쉽고 부적절한 액세스를 허용한다. 파일 및 메모리 보호가 제대로 작동하더라도 데이터가 복사되거나 삭제될 수 있다. 외부 및 내부 공격으로부터 시스템을 보호하는 것은 안전한 작업이다. 이러한 공격 중 일부를 방지하는 것은 일부 시스템에서 운영 체제 기능으로 간주되는 반면, 다른 시스템은 정책이나 기타 소프트웨어에 맡긴다.

1.10. Kernel Data Structures

1.10.1. Lists, Stacks, and Queues

배열은 각 요소에 직접 액세스할 수 있는 간단한 데이터 구조이다. 배열 다음으로 목록은 아마도 컴퓨터 과학에서 가장 기본적인 데이터 구조일 것이다. 배열의 각 항목은 직접 액세스할 수 있지만 목록의 항목은 특정 순서로 액세스해야 한다. 즉, 목록은 데이터 값의 모음을 시퀀스로 나타낸다. 이 구조를 구현하는 가장 일반적인 방법은 항목이 서로 연결된 연결 목록입니다. 연결 목록에는 여러 유형이 있다.
단일 항목 목록에서 각 항목은 후속 항목을 가리킨다.
양방향 목록에서 주어진 항목은 이전 항목과 후속 항목을 모두 참조할 수 있다.
순환 목록에서 연결 목록의 마지막 요소는 null이 아닌 첫 번째 요소를 참조한다.

스택은 항목을 추가하고 제거하기 위해 LIFO원칙을 사용하는 정렬된 데이터 구조이다. 즉, 스택에 마지막으로 배치된 항목이 제거되는 첫 번째 항목이다. 스택에서 항목을 삽입하고 제거하는 작업을 각각 push 및 pop이라고 한다. 운영 체제는 종종 함수를 호출할 때 스택을 사용한다. 인수, 지역 변수 및 반환 주소는 함수가 호출될 때 스택에 푸시되고, 함수 호출에서 반환하면 이러한 항목이 스택에서 꺼진다.

이와 대조적으로 큐는 FIFO원칙을 사용하는 순차적인 데이터 구조이다. 즉, 항목이 삽입된 순서대로 큐에서 제거된다. 계산을 위해 매장에 줄을 선 쇼핑객과 신호등에 줄을 서 있는 자동차 등 매일 대기열이 발생하는 예가 많이 있다.

1.10.2. Trees

트리는 데이터를 계층적으로 표현하는 데 사용할 수 있는 데이터 구조이다. 트리 구조의 데이터 값은 부모-자식 관계로 연결된다. 일반 트리에서 부모는 무한한 수의 자식을 가질 수 있다. 이진 트리에서 부모는 최대 두 개의 자식을 가질 수 있으며 이를 왼쪽 자식과 오른쪽 자식이라고 한다. 이진 검색 트리는 왼쪽 자식 <= 오른쪽 자식인 부모의 두 자식 간에도 정렬해야 한다.

1.10.3. Hash Functions and Maps

해시 함수는 데이터를 입력으로 받아 해당 데이터에 대해 수치 연산을 수행하고 숫자를 반환한다. 그러면 이 값을 테이블(일반적으로 배열)에 대한 인덱스로 사용하여 데이터를 빠르게

검색할 수 있다. 크기가 n인 목록을 통해 데이터 항목을 검색하는 경우 최악의 경우 최대 O(n)개의 비교가 필요할 수 있지만 해시 함수를 사용하여 테이블에서 데이터를 검색하는 것은 최악의 경우 O(1)만큼 좋을 수 있다. 구현 세부 정보에 따라 다르다. 이러한 성능으로 인해 해시 함수는 운영 체제에서 널리 사용된다.

1.10.4. Bitmaps

비트맵은 n개 항목의 상태를 나타내는 데 사용할 수 있는 n개의 이진수 문자열이다. 예를 들어 여러 리소스가 있고 각 리소스의 가용성이 이진수 값으로 표시된다고 가정한다. 0은 리소스를 사용할 수 있음을 의미하고 1은 사용할 수 없음을 의미한다(반대의 경우도 마찬가지). 비트맵의 i번째 위치에 있는 값은 i번째 리소스와 연결된다.

비트맵의 힘은 공간 효율성을 고려할 때 분명해진다. 단일 비트 대신 8비트 부울을 사용하면 결과 데이터 구조가 8배 더 커진다. 따라서 비트맵은 많은 리소스의 가용성을 나타낼 필요가 있을 때 자주 사용된다. 디스크 드라이브는 좋은 설명을 제공한다. 중간 크기의 디스크 드라이브는 디스크 블록이라고 하는 수천 개의 개별 단위로 나눌 수 있다. 비트맵을 사용하여 각 디스크 블록의 가용성을 나타낼 수 있다.

1.11. Computing Environments

1.11.1. Traditional Computing

현재 추세는 이러한 컴퓨팅 환경에 액세스할 수 있는 더 많은 방법을 제공하는 것이다. 웹 기술과 계속 증가하는 WAN 대역폭은 기존 컴퓨팅의 경계를 확장하고 있다. 회사는 내부 서버에 대한 웹 액세스를 제공하는 포털을 구축한다. 기본적으로 웹 기반 컴퓨팅을 이해하는 터미널인 네트워크 컴퓨터(또는 씬 클라이언트)는 더 강력한 보안이 필요하거나 유지 관리가 더 쉬운 기존 워크스테이션을 대체하는 데 사용된다. 모바일 컴퓨터는 PC와 동기화할 수 있으므로 기업 정보를 매우 쉽게 이동할 수 있다. 모바일 컴퓨터는 무선 및 셀룰러 데이터 네트워크에 연결하여 기업 웹 포털(및 기타 수많은 웹 리소스)을 사용할 수도 있다.

집에서 대부분의 사용자는 사무실, 인터넷 또는 둘 다에 모뎀 연결이 느린 컴퓨터를 사용했다. 오늘날 많은 곳에서 한때 높은 비용으로 제공되었던 네트워크 연결 속도가 상대적으로 저렴하여 가정 사용자가 더 많은 데이터에 더 많이 액세스할 수 있다. 이러한 빠른 데이터 연결을 통해 가정용 컴퓨터는 웹 페이지를 제공하고 프린터, 클라이언트 PC 및 서버를 포함한 네트워크를 실행할 수 있다. 많은 가정에서 방화벽을 사용하여 보안 침해로부터 네트워크를 보호한다.

1.11.2. Mobile Computing

모바일 컴퓨팅은 휴대형 스마트폰과 태블릿에서 이루어지는 컴퓨팅을 의미한다. 이러한 장치는 휴대가 간편하고 가볍다는 놀라운 물리적 특성을 가지고 있다.

현재 모바일 컴퓨팅을 지배하는 두 가지 운영 체제는 Apple iOS와 Google Android이다. iOS는 Apple iPhone 및 iPad 모바일 장치에서 실행되도록 설계되었다. Android는 많은 제조업체의 스마트폰과 태블릿을 지원한다.

1.11.3. Distributed Systems

가장 간단한 용어로 네트워크는 둘 이상의 시스템 간의 통신 경로이다. 분산 시스템의 기능은 네트워크에 따라 다르다. 네트워크는 사용되는 프로토콜, 노드 간 거리 및 전송 매체에 따라 다른다. TCP/IP는 가장 일반적인 네트워크 프로토콜이며 인터넷의 기본 아키텍처를 제공한다. 대부분의 운영 체제는 모든 일반 운영 체제를 포함하여 TCP/IP를 지원한다. 일부 시스템은 필요에 맞게 독점 프로토콜을 지원한다. 운영 체제에 관한 한 네트워크 프로토콜은 인터페이스 장치(예: 네트워크 어댑터)와 이를 관리하는 장치 드라이버, 데이터를 처리하는 소프트웨어만 있으면 된다.

네트워크는 노드 간의 거리를 기반으로 특성화된다. 근거리 통신망(LAN)은 방, 건물 또는 캠퍼스의 컴퓨터를 연결한다. WAN(광역 네트워크)은 일반적으로 건물, 도시 또는 국가를 연결한다.

1.11.4. Client –Server Computing

PC가 더 빠르고 강력해지고 저렴해짐에 따라 설계자들은 중앙 집중식 시스템 아키텍처에서 멀어졌다. 중앙 집중식 시스템에 연결된 터미널은 이제 개인용 컴퓨터와 모바일 장치로 대체되고 있다. 따라서 한때 중앙 집중식 시스템에서 직접 처리되던 사용자 인터페이스 기능이 웹 인터페이스를 통해 PC에서 점점 더 많이 처리되고 있다. 따라서 오늘날의 많은 시스템은 클라이언트 시스템에서 생성된 요청을 이행하기 위해 서버 시스템으로 작동한다. 이러한 형태의 특수 분산 시스템을 client-server 시스템이라고 한다.

서버 시스템은 크게 컴퓨팅 서버와 파일 서버로 나눌 수 있다.
컴퓨팅 서버 시스템은 클라이언트가 작업(예: 데이터 읽기)을 수행하기 위한 요청을 보낼 수 있는 인터페이스를 제공한다. 이에 대한 응답으로 서버는 작업을 수행하고 결과를 클라이언트에 보낸다.
파일 서버 시스템은 클라이언트가 파일을 생성, 업데이트, 읽기 및 삭제할 수 있는 파일 시스템 인터페이스를 제공한다.

1.11.5. Peer-to-Peer Computing

분산 시스템의 또 다른 구조는 P2P(Peer-to-Peer) 시스템 모델이다. 이 모델에서는 클라이언트와 서버 사이에 구분이 없다. 대신 시스템 내의 모든 노드는 피어로 간주되며 각 노드는 서비스를 요청하거나 제공하는지 여부에 따라 클라이언트 또는 서버 역할을 할 수 있다. P2P 시스템은 기존 client-server 시스템에 비해 이점을 제공한다. client-server 시스템에서 서버는 병목 지점이지만 피어 투 피어 시스템에서는 네트워크 전체에 분산된 여러 노드에서 서비스를 제공할 수 있다.

1.11.6. Virtualization

가상화는 운영 체제가 다른 운영 체제에서 응용 프로그램으로 실행될 수 있도록 하는 기술이다. 언뜻보기에는 그러한 기능에 대한 이유가 거의 없어 보인다. 그러나 가상화 산업은 방대하고 성장하며 그 유용성과 중요성을 입증하고 있다.

1.11.7. Cloud Computing

클라우드 컴퓨팅은 컴퓨팅, 스토리지 및 애플리케이션을 네트워크를 통해 서비스로 제공하는 컴퓨팅 유형이다. 어떤 면에서는 가상화를 기능의 기반으로 사용하기 때문에 가상화의 논리적 확장이다.
실제로 다음을 포함하여 많은 유형의 클라우드 컴퓨팅이 있다.

- 공용 클라우드 — 서비스 비용을 지불할 의사가 있는 모든 사람이 인터넷을 통해 사용할 수 있는 클라우드
- 프라이빗 클라우드 — 회사가 자체적으로 사용하기 위해 회사에서 운영하는 클라우드
- 하이브리드 클라우드 — 퍼블릭 및 프라이빗 클라우드 구성 요소를 모두 포함하는 클라우드
- SaaS(Software as a Service) — 인터넷을 통해 제공되는 하나 이상의 응용 프로그램(예: 워드 프로세서 또는 스프레드시트)
- PaaS(Platform as a Service) — 인터넷을 통해 애플리케이션에서 사용할 수 있는 소프트웨어 스택(예: 데이터베이스 서버)
- IaaS(Infrastructure as a Service) — 인터넷을 통해 제공되는 서버 또는 스토리지(예: 프로덕션 데이터의 백업 복사본을 만드는 데 사용할 수 있는 스토리지)

1.11.8. Real-Time Embedded Systems

임베디드 컴퓨터는 현존하는 가장 일반적인 컴퓨터 형태이다. 이러한 임베디드 시스템은 매우 다양하다. 일부는 기능을 위한 특수 응용 프로그램과 함께 표준 운영 체제(예: Linux)를 실행하는 범용 컴퓨터이다. 나머지는 필요한 기능만 제공하는 전용 임베디드 운영 체제가 있는 하드웨어 장치이다. 또 다른 것들은 운영 체제 없이 작업을 수행하는 ASIC(주문형 집적 회로)가 있는 하드웨어 장치이다.

임베디드 시스템은 거의 항상 실시간 운영 체제를 실행한다. 실시간 시스템은 프로세서 또는 데이터 흐름의 작동에 엄격한 타이밍 요구 사항이 있을 때 사용되므로 특수 응용 프로그램에서 제어 장치로 자주 사용된다.

실시간 시스템에는 잘 정의된 고정 시간 제약 조건이 있다. 정의된 제약 조건 내에서 처리를 수행해야 하며 그렇지 않으면 시스템이 실패한다.

1.12. Open-Source Operating Systems

1.12.1. History

컴퓨터 및 소프트웨어 회사는 궁극적으로 승인된 컴퓨터와 유료 고객으로 소프트웨어 사용을 제한하려고 한다. 소스 코드 자체가 아닌 소스에서 컴파일된 바이너리만 릴리스하는 것이 이를 달성하고 경쟁자로부터 코드와 아이디어를 보호하는 데 도움이 된다. 또 다른 문제는 저작권이 있는 자료에 관한 것이다. 운영 체제 및 기타 프로그램은 인증된 컴퓨터에서 영화 및 음악을 재생하거나 전자 책을 표시하는 기능을 제한할 수 있다. 이러한 복제 방지 또는 DRM(디지털 권한 관리)은 이러한 제한을 구현하는 소스 코드가 배포되는 경우 효과가 없다. 미국 DMCA(디지털 밀레니엄 저작권법)를 비롯한 많은 국가의 법률에 따라 DRM 코드를 리버스 엔지니어링하거나 복사 방지를 우회하려는 시도가 불법이다.

소프트웨어 사용 및 재배포를 제한하려는 움직임에 대응하여 Richard Stallman은 1983년에 무료 오픈 소스 UNIX 호환 운영 체제를 만들기 위해 GNU 프로젝트를 시작했다. 1985년 그는 모든 소프트웨어가 무료이고 오픈 소스여야 한다고 주장하는 GNU 선언문을 발표했다. 그는 또한 소프트웨어 소스 코드의 자유로운 교환과 해당 소프트웨어의 자유로운 사용을 장려하는 것을 목표로 하는 자유 소프트웨어 재단(FSF)을 설립했다. FSF는 소프트웨어의 저작권을 보호하는 대신 소프트웨어를 "카피레프트"하여 공유 및 개선을 장려한다. 카피레프트는 자유 소프트웨어 배포를 위한 일반 라이선스인 GPL(GNU General Public License)에 의해 성문화되어 있다. 기본적으로 GPL은 소스 코드를 바이너리와 함께 배포해야 하며 소스 코드에 대한 모든 변경 사항은 동일한 GPL 라이선스에 따라 릴리스되어야 한다.

1.12.2. Linux

오픈 소스 운영 체제의 예로 GNU/Linux를 생각해 보십시오. GNU 프로젝트는 컴파일러, 편집기 및 유틸리티를 포함하여 많은 UNIX 호환 도구를 생산했지만 커널을 출시하지는 않았다. 1991년에 핀란드 학생인 Linus Torvalds는 GNU 컴파일러와 도구를 사용하여 기본 UNIX와 유사한 커널을 출시하고 전 세계의 기여자를 초대했다. 인터넷의 출현은 관심 있는 모든 사람이 소스 코드를 다운로드하여 수정하고 Torvalds에 변경 사항을 제출할 수 있음을 의미했다. 주간 업데이트를 통해 소위 Linux 운영 체제가 빠르게 성장할 수 있었고 수천 명의 프로그래머가 향상되었다.

1.12.3. BSD UNIX

Linux와 마찬가지로 FreeBSD, NetBSD, OpenBSD 및 DragonflyBSD를 비롯한 많은 BSD UNIX 배포판이 있다. FreeBSD의 소스 코드를 탐색하려면 관심 있는 버전의 가상 머신 이미지를 다운로드하고 Linux에 대해 위에서 설명한 대로 VMware에서 부팅하기만 하면 된다. 소스 코드는 배포판과 함께 제공되며 /usr/src/에 저장된다. 커널 소스 코드는 /usr/src/sys에 있다. 예를 들어, FreeBSD 커널에서 가상 메모리 구현 코드를 검사하려면 /usr/src/sys/vm에 있는 파일을 보십시오.

1.12.4. Solaris

Solaris는 Sun Microsystems의 상용 UNIX 기반 운영 체제이다. 원래 Sun의 SunOS 운영 체제는 BSD UNIX를 기반으로 했다. Sun은 1991년 AT&T의 System V UNIX를 기반으로

마이그레이션했다. 2005년에 썬은 대부분의 솔라리스 코드를 OpenSolaris 프로젝트로 오픈 소스화했다.

1.12.5. Open-Source Systems as Learning Tools

GNU/Linux와 BSD UNIX는 모두 오픈 소스 운영 체제이지만 각각 고유한 목표, 유틸리티, 라이선스 및 용도가 있다. 때때로 라이선스가 상호 배타적이지 않고 교차 수분이 발생하여 OS 프로젝트를 빠르게 개선할 수 있다.

2. Operating-System Structures

이 장의 목표

- 운영체제가 사용자에게 제공하는 서비스, 프로세스, 다른 시스템 설명
- 운영체제를 구성하는 다양한 방법에 대해 설명
- 운영체제의 설치 및 사용자 지정 방법 및 부팅 방법 설명

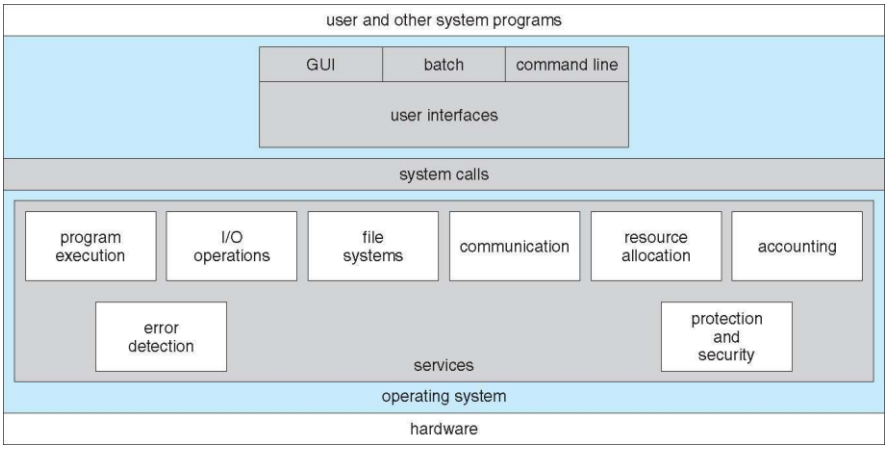
2.1. Operating-System Services

OS는 프로그램 실행환경과 유저, 프로그램에 다음과 같은 서비스를 제공한다.

- User Interface(UI): command-line(CLI-명령창), Graphics User Interface, Batch
- Program execution: 시스템은 정상적으로 또는 비정상적으로 프로그램을 메모리에 로드하고 프로그램을 실행하거나 실행을 종료한다.
- I/O operations: 실행중인 프로그램에는 I/O가 필요할 수 있으며 파일, I/O 장치가 포함될 수 있다.
- File-system manipulation: 파일 시스템은 특히 중요한데, 프로그램은 파일, 디렉토리를 읽고 쓰고 생성, 삭제하고 검색하고 파일 정보를 나열하고 사용 권한 권리가 필요하다. Storage를 I/O 관점에서 생각하는 것이 file system이고 storage는 보통 block device라 얘기하는데, block이라는 벌크 데이터를 handling 한다. 벌크 데이터를 효율적으로 관리하기 위해서 file이 필요하고 엄청난 양의 데이터가 저장소에 있는데 특정한 데이터를 mapping 하기 위해서는 file object가 필요하다. 거대한 저장소의 특정 block을 file의 한 부분으로 처리하는 것이 file system manipulation이다.
- Communication: 프로세스는 동일한 컴퓨터 또는 네트워크를 통해 컴퓨터 간 정보 교환 가능하다.
- Error detection: 운영체제는 발생할 수 있는 오류를 지속적으로 인식할 수 있어야 한다.

프로그램을 잘 실행하는 것이 중요하다.

- Resource Allocation: cpu cycle, main memory, file storage를 효과적으로 배분하는 것 중요
- Accounting: 어떤 사용자가 어떤 종류의 리소스를 사용하고 얼마나 많이 사용하는지 계속해서 추적
- Protection(접근 통제): internal error 방지
- Security(보안): external attack 방지
  →protection과 security handling 중요



[Figure 2.1 A view of operating system services]

프로그램은 UI를 이용해 실행하는데, 실행은 운영체제가 하고 프로그램을 실행해달라고 메시지를 전달하는 방식이 UI이다. UI를 통해 OS에 명령, 서비스를 의뢰할 때 system call을 사용한다.

2.2. User and Operating-System Interface

2.2.1. Command Interpreters

CLI 또는 Command Interpreters(명령 인터프리터)가 직접 명령 입력을 허용한다. 커널에 의해 구현되기도 하고 시스템 프로그램에 의해 구현되기도 한다. shel은 들어온 명령을 해석해서 kernel에 의뢰해주는 접수창고와 같은 역할을 한다. 사용자로부터 명령을 가져와 실행하는데 기본 명령을 제공하기도하고 프로그램 이름만 제공하기도 하는데 시간이 지나면 새로운 기능을 추가하는 데에 shell 수정은 필요하지 않다.

2.2.2. Graphical User Interfaces

운영 체제와의 인터페이스를 위한 두 번째 전략은 사용자 친화적인 그래픽 사용자 인터페이스 또는 GUI를 통해서이다. 여기서 사용자는 명령 줄 인터페이스를 통해 직접 명령을 입력하는 대신, 마우스 기반 창 및 메뉴 시스템을 사용한다. 사용자는 프로그램, 파일, 디렉터리 및 시스템 기능을 나타내는 화면의 이미지 또는 아이콘에 포인터를 배치하기 위해 마우스를 움직이는데, 마우스 포인터의 위치에 따라 마우스 단추를 클릭하면 프로그램을 호출하거나 파일이나 디렉터리를 선택하거나, 명령이 포함된 메뉴를 풀다운할 수 있다.

대부분의 모바일 시스템에서 마우스는 실용적이지 않기 때문에 스마트폰과 휴대용 태블릿 컴퓨터는 일반적으로 터치스크린 인터페이스를 사용한다. 사용자는 터치스크린에서 제스처를 취함으로써 상호 작용하는데 예를 들어 화면을 가로질러 손가락을 누르고 휙휙 움직인다. 그림 2.3은 애플 아이패드의 터치스크린을 보여준다. 초기 스마트폰에 물리적 키보드가 포함된 반면, 대부분의 스마트폰은 이제 터치스크린에서 키보드를 시뮬레이션 한다.

[Figure 2.3 The iPad touchscreen]

2.3. System Calls

시스템 호출은 운영 체제에서 사용할 수 있는 서비스에 대한 인터페이스를 제공한다. 이러한 호출은 일반적으로 C와 C++로 작성된 루틴으로 사용 가능하지만, 특정 낮은 수준의 작업(하드웨어에 직접 액세스해야 하는 작업)은 어셈블리 언어 명령을 사용해야 작성해야 할 수 있다.

운영 체제가 시스템 호출을 사용할 수 있게 하는 방법에 대해 논의하기 전에, 한 파일에서 데이터를 읽고 다른 파일로 복사하는 간단한 프로그램을 작성하는 시스템 호출을 사용하는 예를 들어보겠다.
프로그램이 필요로 하는 첫 번째 입력은 입력 파일과 출력 파일 두 파일의 이름이다. 이러한 이름은 운영 체제 설계에 따라 여러 가지 방법으로 지정할 수 있다.

한 가지 방법은 프로그램이 사용자에게 이름을 묻는 것이다. 대화형 시스템에서 이 접근 방식은 먼저 화면에 프롬프트 메시지를 작성한 다음 키보드에서 두 파일을 정의하는 문자를 읽는 일련의 시스템 호출이 필요하다.

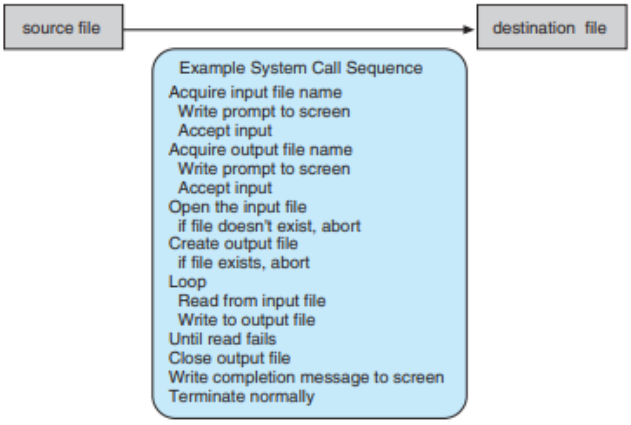
마우스 기반 및 아이콘 기반 시스템에서는 일반적으로 파일 이름의 메뉴가 창에 표시된다. 그런 다음 마우스를 사용하여 소스 이름을 선택할 수 있으며, 대상 이름을 지정할 수 있는 창이 열린다. 이 시퀀스에는 많은 I/O 시스템 호출이 필요하다.

두 개의 파일 이름을 얻으면 프로그램은 입력 파일을 열고 출력 파일을 만들어야 한다. 각 작업에는 다른 시스템 호출이 필요하고 각 작업에 대해 가능한 오류 조건은 추가 시스템 호출이 필요할 수 있다. 예를 들어 프로그램이 입력 파일을 열려고 할 때 해당 이름의 파일이 없거나 해당 파일이 액세스로부터 보호되는 것을 발견할 수 있다. 이 경우 프로그램은 콘솔에 메시지를 출력한다음 비정상적으로 종료된다.

입력 파일이 있으면 새 출력 파일을 만들어야 하는데, 동일한 이름의 출력 파일이 이미 있을 수도 있다. 이 경우 프로그램이 중단되거나(시스템 호출), 기존 파일(다른 시스템 호출)을 삭제하고 새 파일(다른 시스템 호출)을 만들 수 있다. 대화형 시스템에서 다른 옵션은 사용자에게 기존 파일을 교체할지 프로그램을 중단할지 여부를 묻는 것이다.

두 파일이 모두 설정되면 입력 파일에서 읽고(시스템 호출) 출력 파일에 쓰는 루프를 입력한다(다른 시스템 호출). 각 읽기 및 쓰기는 다양한 가능한 오류 조건에 대한 상태 정보를 반환해야 하고 입력 시 프로그램은 파일의 끝에 도달했거나 읽기에 하드웨어 오류가 있음을 발견할 수 있다. (예: 패리티 오류) 쓰기 작업도 출력 장치에 따라 다양한 오류가 발생할 수 있다. (예: 더 이상의 디스크 공간 없음)

마지막으로, 전체 파일이 복사된 후 프로그램은 두 파일을 모두 닫을 수 있다. 콘솔이나 창에 메시지를 쓰고 난 후 정상적으로 종료된다.(최종 시스템 호출) 이 시스템 호출 시퀀스는 그림 2.5에 나와 있다.



[Figure 2.5 Example of how system calls are used]

Figure 2.5를 보면, 간단한 프로그램도 운영 체제를 많이 사용할 수 있다. 시스템은 종종 초당 수천 개의 시스템 호출을 실행한다. 그러나 대부분의 프로그래머들은 이런 수준의 세부사항을 보지 못한다.

일반적으로 응용 프로그램 개발자는 응용 프로그램 프로그래밍 인터페이스(API)에 따라 프로그램을 설계한다. API는 응용 프로그램 프로그래머가 사용할 수 있는 함수 집합을 지정하는데, 여기에는 각 함수에 전달되는 매개 변수와 프로그래머가 기대할 수 있는 반환 값이 포함된다. 응용 프로그램 프로그래머들이 사용할 수 있는 가장 일반적인 세 가지 API는 윈도우

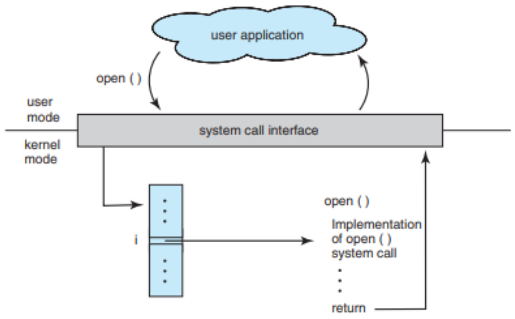
시스템용 윈도우 API, POSIX 기반 시스템용 POSIX API, 자바 가상 머신에서 실행되는 프로그램용 자바 API이다. 프로그래머는 운영 체제에서 제공하는 코드 라이브러리를 통해 API에 액세스한다. C 언어로 작성된 프로그램을 위한 유닉스 및 리눅스의 경우 라이브러리를 libc라고 한다.

애플리케이션 프로그래머가 실제 시스템 호출을 호출하는 것보다 API에 따른 프로그래밍을 선호하는 데에는 몇 가지 이유가 있다. 한 가지 이점은 프로그램 이동성과 관련이 있다. API를 사용하여 프로그램을 설계하는 응용 프로그램 프로그래머는 자신의 프로그램이 동일한 API를 지원하는 모든 시스템에서 컴파일 및 실행을 기대할 수 있다. 게다가, 실제 시스템 호출은 종종 애플리케이션 프로그래머가 사용할 수 있는 API보다 더 상세하고 작업하기가 어려울 수 있다. 그럼에도 불구하고 API의 함수와 커널 내의 관련 시스템 호출 사이에는 강한 상관관계가 존재한다. 실제로 POSIX와 윈도우 API의 대부분은 유닉스, 리눅스, 윈도우 운영 체제에서 제공하는 네이티브 시스템 호출과 유사하다.

대부분의 프로그래밍 언어에서 런타임 지원 시스템(컴파일러와 함께 라이브러리에 내장된 함수 집합)은 운영 체제에서 사용할 수 있는 시스템 호출에 대한 링크 역할을 하는 시스템 호출 인터페이스를 제공한다. 시스템 호출 인터페이스는 API에서 함수 호출을 가로채고 운영 체제 내에서 필요한 시스템 호출을 호출한다. 일반적으로 번호는 각 시스템 호출과 연결되며, 시스템 호출 인터페이스는 이 번호에 따라 인덱스된 테이블을 유지한다.

그런 다음 시스템 호출 인터페이스는 운영 체제 커널에서 의도된 시스템 호출을 호출하고 시스템 호출의 상태와 반환 값을 반환한다. 호출자는 시스템 호출이 어떻게 구현되는지 또는 실행 중에 무엇을 하는지 전혀 알 필요가 없다. 오히려, 호출자는 API를 따르고 운영 체제가 그 시스템 호출의 실행의 결과로 무엇을 할 것인지 이해하기만 하면 된다.

따라서 운영 체제 인터페이스의 대부분의 세부 사항은 API에 의해 프로그래머에게 숨겨지고 런타임 지원 라이브러리에 의해 관리된다. API, 시스템 호출 인터페이스 및 운영 체제 간의 관계는 Figure 2.6에 나타나 있으며, 이는 운영 체제가 open() 시스템 호출을 호출하는 사용자 애플리케이션을 처리하는 방법을 보여준다.



[Figure 2.6 The handling of a user application invoking the open() system call]

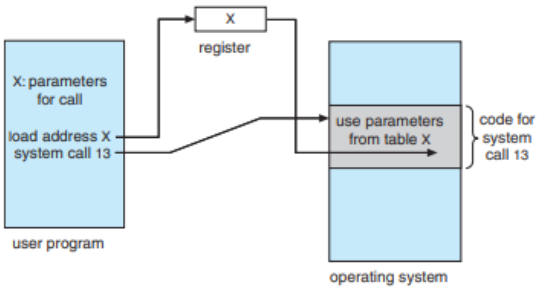
시스템 호출은 사용 중인 컴퓨터에 따라 다른 방식으로 발생하고 종종 원하는 시스템 호출의 ID보다 더 많은 정보가 필요하다. 정확한 정보 유형과 양은 특정 운영 체제 및 호출에 따라

다르다. 예를 들어, 입력을 받기 위해 소스로 사용할 파일이나 장치뿐만 아니라 입력을 읽을 메모리 버퍼의 주소 및 길이를 지정해야 할 수 있다.

시스템 호출을 하다 보면 여러 파라미터를 전달해야 할 때가 있다. 파라미터를 운영체제에 전달하는 방법으로는 크게 3가지가 있다.

첫 번째로는 원하는 값을 레지스터에 담아서 전달하는 가장 간단한 방법이다. 그러나 이 방법은 경우에 따라 레지스터보다 더 많은 매개 변수가 있을 수 있으므로 잘 사용하지 않는다.

두 번째로는 파라미터를 block, table, memory에 넣어두고 포인터를 시스템 호출에 전달하는 방법이다. 이 방법은 Call by Reference로 많은 양을 전달하기에 좋고 포인터를 함수의 인자로 옮겨주는 방식으로, 포인터를 넘겨줌으로써 함수호출의 overhead를 줄여준다는 장점이 있다. 세 번째로는 kernel과 사용자 process가 stack을 공유하는 방법인데 이는 stack에 pop, push등을 하는 것으로 Call by Value 방식이다. 복사를 해야만 한다는 특징이 있어 시공간적 문제가 발생한다는 단점이 있다.



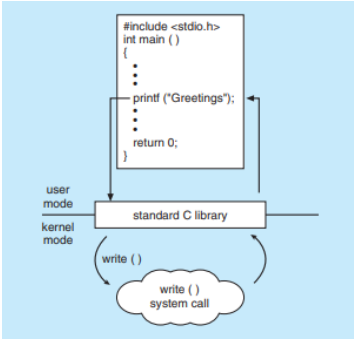
[Figure 2.7 Passing of parameters as a table]

2.4. Types of System Calls

시스템 호출은 크게 process control, file manipulation, device manipulation, information maintenance, communications, and protection의 6가지 주요 범주로 분류할 수 있다. Unix의 시스템 호출을 살펴보면, Process Control에는 fork() & exit() & wait(), File Manipulation에는 open() & read() & write() & close(), Device Manipulation에는 ioctl() & read() & write(), Information Maintenance에는 getpid() & alarm() & sleep(), Communication에는 pipe() & shm_open() & mmap(), Protection에는 chmod() & umask() & chown()이 있다.

표준 C 라이브러리는 많은 버전의 유닉스 및 리눅스용 시스템 호출 인터페이스의 일부를 제공한다.

예를 들어, C 프로그램이 printf() 문을 호출한다고 가정하자. C 라이브러리는 이 호출을 가로채 운영 체제에서 필요한 시스템 호출을 호출하고C 라이브러리는 반환된 값을 가져온다. 그 후, write road를 다시 작성하여 사용자 프로그램으로 전달한다.

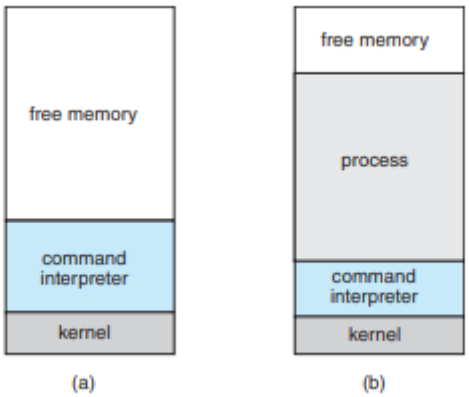


[EXAMPLE OF STANDARD C LIBRARY]

프로세스 및 작업 제어에는 다양한 측면이 있다. 단일 작업 시스템과 멀티 작업 시스템을 포함하는 두 가지 예를 사용하여 이러한 개념을 명확히 이해할 수 있다.

MS-DOS 운영 체제는 단일 작업 시스템의 한 예이다. 컴퓨터를 시작할 때 호출되는 명령 인터프리터가 있다. (그림 a) MS-DOS는 단일 작업이기 때문에 프로그램을 실행하기 위해 간단한 방법을 사용하며 새로운 프로세스를 만들지 않는다.

프로그램을 메모리에 로드하고, 프로그램에 가능한 많은 메모리를 제공하기 위해 스스로 덮어쓴다. (그림 b) 다음으로 명령 포인터를 프로그램의 첫 번째 명령으로 설정한다. 그 후에 프로그램이 실행되고 오류가 트랩을 발생시키거나 프로그램이 종료하기 위해 시스템 호출을 실행한다. 두 경우 모두 나중에 사용할 수 있도록 오류 코드가 시스템 메모리에 저장된다. 이 작업을 수행한 후 덮어쓰지 않은 명령 인터프리터의 작은 부분이 실행을 재개한다. 첫 번째 작업은 명령 인터프리터의 나머지 부분을 디스크에서 다시 로드하는 것이다. 그런 다음 명령 인터프리터는 이전 오류 코드를 사용자 또는 다음 프로그램에서 사용할 수 있도록 한다.



[Figure 2.9 MS-DOS execution. (a) At system startup. (b) Running a program]

2.5. Operating-System Design and Implementation

이 섹션에서는 운영 체제를 설계하고 구현할 때 직면하는 문제에 대해 논의한다. 물론 그러한 문제에 대한 완전한 해결책은 없지만, 성공적인 접근법이 있다.

2.5.1. Design Goals

시스템을 설계할 때 첫 번째 문제는 목표와 사양을 정의하는 것이다. 요구사항은 사용자 목표와 시스템 목표라는 두 가지 기본 그룹으로 나눌 수 있다.

사용자는 시스템에서 특정 명백한 속성을 원한다. 사용의 편리성, 배움의 쉬움, 사용의 쉬움, 신뢰성, 안전성, 빠른 속도 등을 원한다. 물론, 이러한 것들은 달성하는 방법에 대한 일반적인 합의가 없기 때문에 시스템 설계에서 특별히 유용하지는 않다.

시스템 입장에서는 설계, 구현 및 유지보수가 용이해야 하며, 유연하고, 신뢰할 수 있어야 하며, 오류가 없어야 하며, 효율적이어야 한다. 이러한 요구사항은 모호하며 다양한 방식으로 해석될 수 있다.
간단히 말해서, 운영 체제에 대한 요구 사항을 정의하는 문제에 대한 고유한 해결책은 없다. 현존하는 광범위한 시스템은 서로 다른 요구사항이 서로 다른 환경에 대한 다양한 솔루션을 제공할 수 있음을 보여준다.

2.5.2. Mechanisms and Policies

한 가지 중요한 원칙은 정책과 메커니즘을 분리하는 것이다. 메커니즘은 어떤 일을 하는 방법을 결정하며, 정책은 무엇을 할 것인지를 결정한다. 정책과 메커니즘의 분리는 유연성을 위해 중요하다. 정책은 장소에 따라 또는 시간이 지남에 따라 변경될 수 있다. OS를 지정하고 설계하는 것은 소프트웨어 엔지니어링의 창의적인 작업이라고 할 수 있다.

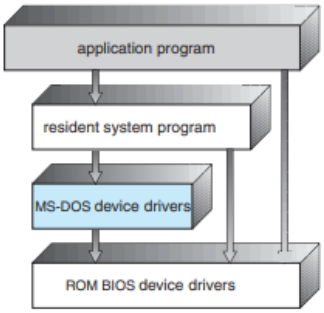
2.6. Operating-System Structure

현대의 운영 체제만큼 크고 복잡한 시스템은 제대로 작동하고 쉽게 수정되려면 신중하게 설계되어야 한다. 일반적인 접근법은 하나의 단일 시스템이 아닌 작은 구성 요소 또는 모듈로 작업을 분할하는 것이다. 각 모듈은 입력, 출력 및 기능이 신중하게 정의되어야 한다. 이 섹션에서는 이러한 구성 요소가 상호 연결되고 커널에 병합되는 방법에 대해 알 수 있다.

2.6.1. Simple Structure

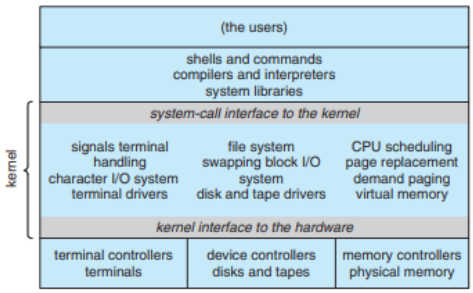
많은 운영 체제들은 잘 정의된 구조를 가지고 있지 않다. 이러한 시스템은 소규모, 단순, 제한된 시스템으로 시작하여 원래의 범위를 넘어서는 경우가 많았다. MS-DOS는 그러한 시스템의 한 예이다. MS-DOS는 원래 그렇게 인기를 끌 것이라는 것을 전혀 모르는 소수의 사람들에 의해

고안되고 실행되었다. 최소 공간에서 가장 많은 기능을 제공하기 위해 작성되었기 때문에 모듈로 신중하게 구분되지 않았다.



[Figure 2.11 MS-DOS layer structure]

MS-DOS에서는 인터페이스와 기능 수준이 잘 분리되어 있지 않다. 이는MS-DOS가 잘못된 프로그램에 취약하게 만들어 사용자 프로그램이 실패할 때 전체 시스템 충돌을 일으킨다.
제한된 구조의 또 다른 예는 원래의 유닉스 운영 체제이다. MS-DOS와 마찬가지로 유닉스도 처음에는 하드웨어 기능에 의해 제한되었다. 커널과 시스템 프로그램이라는 두 개의 분리 가능한 부분으로 구성되어 있다.



[Figure 2.12 Traditional UNIX system structure]

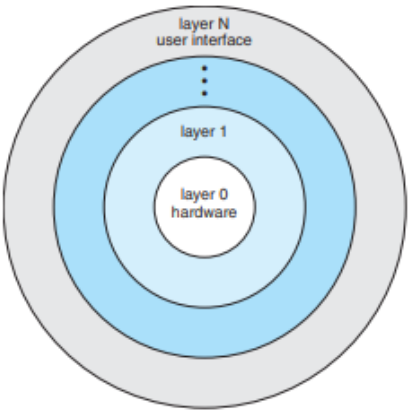
커널은 일련의 인터페이스와 장치 드라이버로 더 분리되어 있으며, 유닉스가 발전함에 따라 수년간 추가되고 확장되었다. 그림 2.12와 같이 전통적인 유닉스 운영 체제를 어느 정도 계층화했다고 볼 수 있다. 시스템 호출 인터페이스 아래의 모든 것과 물리적 하드웨어 위의 모든 것이 커널이다. 커널은 시스템 호출을 통해 파일 시스템, CPU 스케줄링, 메모리 관리 및 기타 운영 체제 기능을 제공한다.
이 획일적인 구조는 구현하고 유지하기가 어려웠으나 시스템 호출 인터페이스나 커널 내 통신에서 오버헤드가 거의 없다는 뚜렷한 성능 이점을 가지고 있었다.

2.6.2. Layered Approach

운영 체제 계층은 데이터와 이러한 데이터를 조작할 수 있는 연산으로 구성된 추상 객체의 구현이다. 일반적인 운영 체제 계층(예: 계층 M)은 데이터 구조와 상위 계층이 호출할 수 있는 일련의 루틴으로 구성된다. 계층 M은 하위 계층에 대한 작업을 호출할 수 있고 계층화된 접근 방식의 주요 장점은 구축과 디버깅의 단순성이다. 각 계층은 하위 계층의 기능(작업)과 서비스만 사용하도록 선택된다.

이 접근 방식은 디버깅 및 시스템 검증을 단순화한다. 첫 번째 계층은 시스템의 나머지 부분에 대한 우려 없이 디버깅될 수 있는데, 이는 정의에 따라 기능을 구현하기 위해 기본 하드웨어만 사용하기 때문이다. 일단 첫 번째 계층이 디버깅되면, 두 번째 계층이 디버깅되는 동안 그것의 정확한 기능을 가정할 수 있다. 특정 계층을 디버깅하는 동안 오류가 발견되면 해당 계층 아래에 있는 계층이 이미 디버깅되어 있기 때문에 해당 계층에 오류가 있음을 알 수 있다.

따라서, 시스템의 설계와 구현이 단순하다. 각 계층은 하위 계층에 의해 제공되는 연산을 통해서만 구현되고 계층은 이러한 작업이 어떻게 구현되는지 알 필요가 없고 무엇을 하는 지만 알면 된다. 따라서 각 계층은 상위 계층으로부터 특정 데이터 구조, 운영 및 하드웨어의 존재를 숨긴다.
계층화 접근법의 주요 어려움은 다양한 계층을 적절하게 정의하는 것이다. 계층은 하위 계층만 사용할 수 있으므로, 신중한 계획이 필요하다. 예를 들어 메모리 관리에는 백업 저장소를 사용할 수 있는 기능이 필요하기 때문에 백업 저장소용 장치 드라이버(가상 메모리 알고리즘에서 사용하는 디스크 공간)는 메모리 관리 루틴보다 낮은 레벨에 있어야 한다.



[Figure 2.13 A layered operating system]

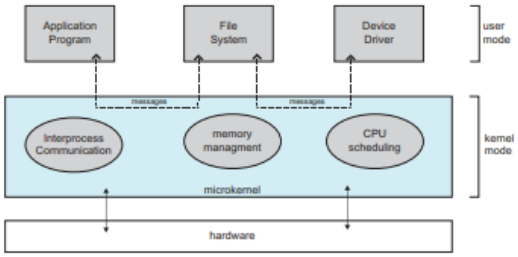
2.6.3. Microkernels

유닉스가 확장되면서 커널이 커지고 관리하기 어렵다는 것을 이미 보았다. 1980년대 중반, 카네기 멜론 대학의 연구원들은 microkernel 접근법을 사용하여 커널을 모듈화하는 운영 체제를 개발하였다. 이는 커널에서 중요하지 않은 모든 구성 요소를 제거하고 시스템 및 사용자 수준 프로그램으로 구현함으로써 운영 체제를 구성한다. 그 결과는 더 작은 커널이다.

어떤 서비스가 커널에 남아 있어야 하고 어떤 서비스가 사용자 공간에서 구현되어야 하는지에 대한 합의가 거의 없다. 그러나 일반적으로 microkernel은 최소한의 프로세스와 메모리 관리를 제공하며 통신 기능도 제공한다. Microkernel의 주요 기능은 클라이언트 프로그램과 사용자 공간에서 실행 중인 다양한 서비스 간의 통신을 제공하는 것이다.

Microkernel 접근법의 한 가지 이점은 운영 체제를 더 쉽게 확장할 수 있다는 것이다. 모든 새 서비스가 사용자 공간에 추가되므로 커널을 수정할 필요가 없다. 커널을 수정해야 할 때, microkernel이 더 작은 커널이기 때문에 변경사항이 더 적은 경향이 있다. 따라서 하드웨어

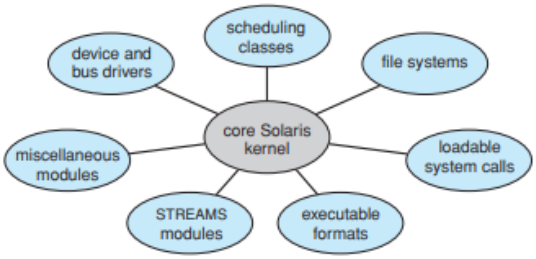
설계 간에 운영 체제를 쉽게 이식할 수 있다. 또한 microkernel은 더 많은 보안을 제공한다. 대부분의 서비스가 커널이 아닌 사용자 프로세스로 실행되기 때문에 안정성이 향상된다. 또한 만약 서비스가 실패해도 나머지 운영 체제는 그대로 유지된다. 하지만 커널 공간 통신에 대한 유저 공간의 성능이 overhead된다는 단점이 있다.



[Figure 2.14 Architecture of a typical microkernel]

2.6.4. Modules

현재 운영 체제 설계를 위한 최선의 방법론은 로드 가능한 커널 모듈을 사용하는 것이고 대부분의 현대 운영 체제는 이를 따르고 있다. 로드 가능한 커널 모듈을 구현한다는 것은 객체 지향 접근 방식 사용, 각 핵심 구성요소 분리, 각각은 알려진 인터페이스를 통해 서로 소통, 각각은 커널 내에서 필요에 따라 로드할 수 있음을 말한다. 이러한 유형의 설계는 솔라리스, 리눅스, 맥 OS X와 같은 유닉스의 현대 구현체 및 윈도우에서 흔히 볼 수 있다. 그림 2.15에 나와 있는 Solaris 운영 체제 구조는 로드 가능한 7가지 유형의 커널 모듈로 구성된 코어 커널을 중심으로 구성되어 있다.



[Figure 2.15 Solaris loadable modules]

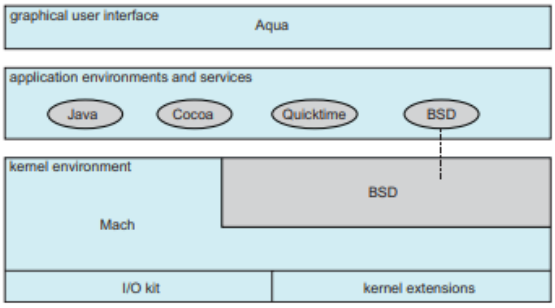
2.6.5. Hybrid Systems

대부분의 현대 운영체제는 사실 하나의 순수한 모델이 아니라 하이브리드 시스템이다. 하이브리드 시스템은 성능, 보안, 사용 편의성 요구를 해결하기 위해 여러 가지 접근 방식을 결합한다.

2.6.5.1. Mac OS X

애플의 MAC OS X 운영체제는 하이브리드 구조이고 계층형 시스템이다. 계층의 상위는 Aqua라는 GUI를 포함하고 있고 어플리케이션 환경 및 서비스에는 Java, Cocoa, Quicktime, BSD 등이 있다. 이 중, 코코아 환경은 맥 OS X 응용 프로그램을 작성하는 데 사용되는 목표 C

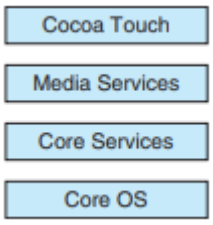
프로그래밍 언어를 위한 API를 지정한다. 이 계층들 아래에는 커널 환경이 있는데, 주로 마하 마이크로 커널과 BSD 유닉스 커널로 구성되어 있다. 마하는 메모리 관리, 메시지 전달을 포함한 원격 프로시저 호출(RPC) 및 프로세스 간 통신(IPC) 기능 지원, 스레드 스케줄링을 제공한다. BSD 컴포넌트는 BSD 명령줄 인터페이스, 네트워킹 및 파일 시스템 지원, Pthreads를 포함한 POSIX API의 구현을 제공한다. 마하와 BSD 외에도 커널 환경은 장치 드라이버와 동적으로 로드 가능한 모듈(맥 OS X가 커널 확장이라고 지칭)을 개발하기 위한 I/O 키트를 제공한다. 그림 2.16과 같이 BSD 애플리케이션 환경은 BSD 설비를 직접 활용할 수 있다.



[Figure 2.16 The Mac OS X structure]

2.6.5.2. iOS

iOS는 애플이 자사의 스마트폰인 아이폰과 태블릿 컴퓨터인 아이패드를 구동하기 위해 설계한 모바일 운영 체제이다. iOS의 구조는 그림 2.17에 나타난다. Cocoa Touch는 iOS 기기에서 실행되는 애플리케이션을 개발하기 위한 여러 프레임워크를 제공하는 Objective-C용 API이다. 앞서 언급한 코코아와 코코아 터치의 근본적인 차이점은 후자가 터치 스크린과 같은 모바일 장치에 고유한 하드웨어 기능을 지원한다는 것이다. Media Services layer는 그래픽, 오디오 및 비디오에 대한 서비스를 제공한다. Core Services layer는 클라우드 컴퓨팅 및 데이터베이스 지원을 비롯한 다양한 기능을 제공한다. Core OS는 그림 2.16에 표시된 커널 환경을 기반으로 하는 핵심 운영 체제를 나타낸다.

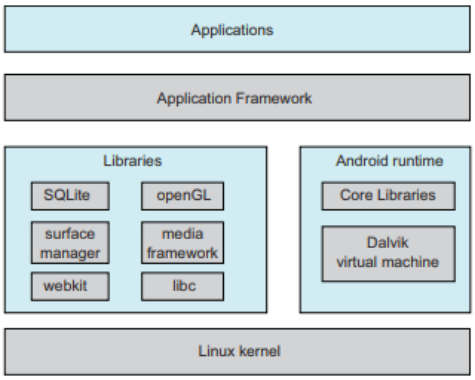


[Figure 2.17 Architecture of Apple's iOS]

2.6.5.3. Android

안드로이드 운영 Open Handset Alliance에 의해 설계되었으며 안드로이드 스마트폰과 태블릿 컴퓨터용으로 개발되었다. iOS가 애플 모바일 기기에서 실행되도록 설계되고 근접 소싱된 반면, 안드로이드는 다양한 모바일 플랫폼에서 실행되며 오픈 소싱으로 빠른 인기 상승을 부분적으로 설명한다. 안드로이드의 구조는 그림 2.18과 같다.

안드로이드는 모바일 애플리케이션 개발을 위한 풍부한 프레임워크 세트를 제공하는 소프트웨어 계층형 스택이라는 점에서 iOS와 유사하다. 이 소프트웨어 스택의 맨 아래에는 리눅스 커널이 있지만, 구글에 의해 수정되었으며 현재 리눅스 릴리스의 정상적인 배포판 밖에 있다. 리눅스는 주로 하드웨어에 대한 프로세스, 메모리, 장치 드라이버 지원을 위해 사용되며 전원 관리를 포함하도록 확장되었고 안드로이드 런타임 환경은 달빅 가상 머신뿐만 아니라 라이브러리의 핵심 집합을 포함한다. 안드로이드 기기용 소프트웨어 설계자들은 자바 언어로 응용 프로그램을 개발한다. 그러나 구글은 표준 자바 API를 사용하는 대신 자바 개발을 위한 별도의 안드로이드 API를 설계했다. 자바 클래스 파일은 먼저 자바바이트코드로 컴파일된 다음 달빅 가상 머신에서 실행되는 실행 파일로 변환된다. 달빅 가상 머신은 안드로이드용으로 설계되었으며 메모리와 CPU 처리 능력이 제한된 모바일 기기에 최적화되어 있다.



[Figure 2.18 Architecture of Google's Android]

2.7. Operating-System Debugging

2.7.1. Failure Analysis

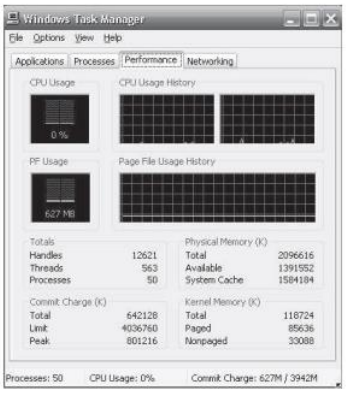
디버깅이란 에러나 버그를 찾고 고치는 작업을 말한다. 프로세스가 실패하면 대부분의 운영 체제는 오류 정보를 로그 파일에 기록하여 시스템 운영자나 사용자에게 문제가 발생했음을 알린다. 운영 체제는 또한 코어 덤프(프로세스의 메모리 캡처)를 가져와 나중에 분석할 수 있도록 파일에 저장할 수 있다. 실행 중인 프로그램과 코어 덤프는 디버거에 의해 탐색 될 수 있으며, 이를 통해 프로그래머는 프로세스의 코드와 메모리를 탐색할 수 있다. 하지만 사용자 수준 프로세스 코드를 디버깅하는 것은 어려운 일이고, 운영 체제 커널 디버깅은 커널의 크기와 복잡성, 하드웨어 제어, 사용자 수준 디버깅 툴의 부족으로 인해 훨씬 더 복잡하다.

커널의 장애는 crash라고 부르고, crash가 발생하면 오류 정보가 로그 파일에 저장되고 메모리 상태가 crash dump에 저장된다. Crash를 넘어서, 성능 tuning을 통해 시스템을 최적화할 수 있다. 분석을 위해 기록된 활동의 추적 목록을 사용하기도 하고 프로파일링을 사용하기도 하는데, 프로파일링이란 통계적 경향을 찾기 위한 명령 포인터의 주기적인 샘플링이다. Kernighan's Law는 "초반에는 코드를 적는 것보다 디버깅이 2배나 더 어렵다. 그래서 만약

코드를 창의적으로 적으려고 한다면, 당신은 말그대로 디버깅에 있어서만큼은 창의적이지 않은 것이다" 라고 말하는데, 이를 통해 디버깅이 어려운 작업임을 알 수 있다.

2.7.2. Performance Tuning

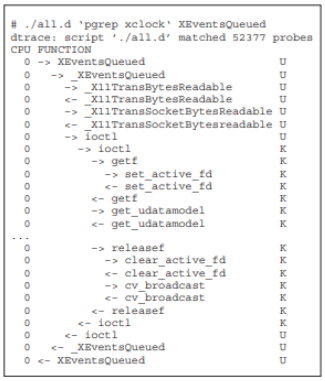
앞서 Performance Tuning은 bottlenecks를 제거하여 성능을 개선하고자 한다고 언급했다. Bottleneck 현상을 식별하려면 시스템 성능을 모니터링할 수 있어야 한다. 그러므로 운영 체제는 시스템 동작의 측정값을 계산하고 표시하는 어떤 수단을 가져야만 한다. 수많은 시스템에서 운영 체제는 시스템 동작의 추적 목록을 생성하여 이를 수행한다. 모든 이벤트는 시간과 중요한 매개 변수로 기록되고 파일에 기록되고 나중에 성능 분석 프로그램은 로그 파일을 처리해 시스템 성능을 확인하고 bottleneck 현상 및 비효율성을 식별할 수 있다. 이러한 동일한 추적을 제안된 개선된 시스템의 시뮬레이션에 대한 입력으로 실행할 수 있고 추적을 통해 운영 체제 동작에서 오류를 찾을 수 있다. 그림 2.19는 Windows 작업 관리자 스크린샷으로, Windows 시스템을 위한 도구이다. 작업 관리자에는 프로세스, CPU 및 메모리 사용량, 네트워킹 통계 뿐만 아니라 현재 애플리케이션에 대한 정보도 포함된다. 운영 체제가 실행되는 동안 운영 체제를 더 쉽게 이해하고 디버깅하고 튜닝할 수 있도록 연구가 필요하다.



[Figure 2.19 The Windows task manager]

2.7.3. DTrace

DTrace는 실행 중인 시스템에 사용자 프로세스와 커널 모두에서 동적으로 probe를 추가하는 기능이다. 이러한 probe는 D 프로그래밍 언어를 통해 질의되어 커널, 시스템 상태 및 프로세스 활동에 대한 놀라운 양을 결정할 수 있다. 예를 들어, 2.20은 응용 프로그램이 시스템 호출(ioctl()을 실행할 때 응용 프로그램을 따르고 커널 내의 기능 호출이 시스템 호출 수행을 위해 실행될 때 보여준다. U로 끝나는 행은 사용자 모드에서 실행되며, K로 끝나는 행은 커널 모드에서 실행된다.



[Figure 2.20 Solaris 10 dtrace follows a system call within the kernel]

2.8. Operating-System Generation

운영 체제는 모든 종류의 기계에서 실행되도록 설계되었고 시스템은 각 특정 컴퓨터 사이트에 대해 구성되어야 한다.

SYSGEN 프로그램은 하드웨어 시스템의 특정 구성에 관한 정보를 얻는다. 이는 시스템별 컴파일 된 커널 또는 시스템 커널을 빌드하는 데 사용되고 하나의 일반적인 커널보다 더 효율적인 코드를 만들 수 있다.

2.9. System Boot

시스템에서 전원이 초기화되면 고정된 메모리 위치에서 실행이 시작되고 펌웨어 ROM은 초기 부팅 코드를 보관하는 데 사용된다. 하드웨어에서 시작할 수 있도록 운영 체제가 하드웨어에서 사용 가능해야만 하고 ROM 또는 EEPROM에 저장된 작은 코드 부트스트랩 로더 조각은 커널을 찾아 메모리에 로드하고 시작한다.

공통 부트스트랩인 GRUB를 사용하면 여러 디스크, 버전, 커널 옵션에서 커널을 선택할 수 있다. 커널이 로드되어야만 시스템이 그 다음에 작동할 수 있다.

3. Processes

오늘날의 컴퓨터 시스템들은 메모리에 다수의 프로그램이 적재되어 병행 실행 되는 것을 허용한다. 이러한 발전은 다양한 프로그램을 보다 견고하게 제어하고 보다 구획화할 것을 필요로 했다. 이러한 필요성이 프로세스의 개념을 낳았으며, 프로세스란 실행 중인 프로그램을 말한다. 프로세스는 현대의 컴퓨팅 시스템에서 작업의 단위이다.

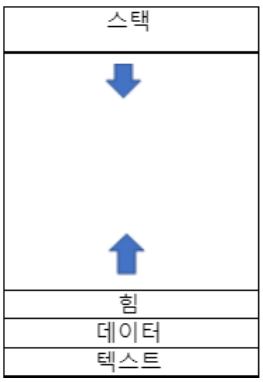
시스템은 일부는 사용자 코드를 실행하고 일부는 운영체제 코드를 실행하는 프로세스의 집합체이다. 이들 모든 프로세스는 잠재적으로 병행 실행이 가능하고,CPU(또는 CPU들)는 이들 프로세스 가운데서 다중화(multiplex)된다. 이 장에서는 프로세스가 무엇인지, 운영체제에서 어떻게 표현되는지 그리고 어떻게 작동하는지에 대해 설명한다.

3.1. Process Concept

우리가 운영체제를 논의할 때 한 가지 장애는 모든 CPU 활동들을 어떻게 부를 것인가하는 데에 의문이 있다는 것이다. 초창기 컴퓨터는 작업(job)을 실행하는 일괄처리 시스템이었고, 사용자 프로그램 또는 태스크(task)를 실행하는 시분할 시스템이 뒤를 이었다. 단일 사용자 시스템에서도 사용자는 워드 프로세서, 웹 브라우저 및 전자 메일 패키지와 같은 여러 프로그램을 한 번에 실행할 수 있다. 또한 다중 태스킹을 지원하지 않는임베디드 장치에서와 같이 컴퓨터가 한 번에 하나의 프로그램만 실행할 수 있더라도 운영체제는 메모리 관리와 같은 자체 프로그램된 내부 활동을 지원해야 할 수도 있다. 여러 측면에서 이러한 모든 활동은 유사하므로 이러한 모든 활동을 프로세스라고 부를 수있다.

3.1.1. The Process

비공식적으로, 프로세스란 실행 중인 프로그램이다. 프로세스의 현재 활동의 상태는 프로그램 카운터 값과 프로세서 레지스터의 내용으로 나타낸다.



1. 텍스트 섹션 – 실행 코드
2. 데이터 섹션 – 전역 변수
3. 힙 섹션 – 프로그램 실행 중에 동적으로 할당되는 메모리
4. 스택 섹션 – 함수를 호출할 때 임시 데이터 저장장소 (예: 함수 매개변수, 복귀 주소및 지역 변수)

텍스트 및 데이터 색션의 크기는 고정되기 때문에 프로그램 실행 시간 동안 크기가변하지 않는다 .
우리는 프로그램 그 자체는 프로세스가 아님을 강조한다. 프로그램은 명령어 리스트를 내용으로 가진 디스크에 저장된 파일(실행 파일이라고 불림)과 같은 수동적인 존재(passive entity)이다. 두 프로세스들이 동일한 프로그램에 연관될 수 있지만, 이들은 두 개의 별도의 실행순서로 간주한다

3.1.2. Process State

프로세스는 실행되면서 그 상태가 변한다. 프로세스의 상태는 부분적으로 그 프로세스의 현재의 활동에 따라서 정의된다. 프로세스는 다음 상태 중 하나에 있게 된다.

1. 새로운(new): 프로세스가 생성 중이다.
2. 실행(running): 명령어들이 실행되고 있다.
3. 대기(waiting): 프로세스가 어떤 이벤트(입출력 완료 또는 신호의 수신 같은)가 일어나기를 기다린다.
4. 준비(ready): 프로세스가 처리기에 할당되기를 기다린다.
5. 종료(terminated): 프로세스의 실행이 종료되었다.

3.1.3. Process Control Block

프로세스 제어 블록은 특정 프로세스와 연관된 여러 정보를 수록하며, 다음과 같은 것들을 포함한다. 프로세스 상태: 상태는 새로운(new), 준비(ready), 실행(running), 대기(waiting)또는 정지(halted) 상태 등이다.

1. 프로그램 카운터: 프로그램 카운터는 이 프로세스가 다음에 실행할 명령어의 주소를가리킨다.
2. CPU 레지스터들: CPU 레지스터는 컴퓨터의 구조에 따라 다양한 수와 유형을 가진다.
3. CPU-스케줄링 정보: 이 정보는 프로세스 우선순위, 스케줄 큐에 대한 포인터와 다른 스케줄 매개변수를 포함한다
4. 메모리 관리 정보: 이 정보는 운영체제에 의해 사용되는 메모리 시스템에 따라 기준(base) 레지스터와 한계(limit) 레지스터의 값, 운영체제가 사용하는 메모리 시스템에 따라 페이지 테이블 또는 세그먼트 테이블 등과 같은 정보를 포함한다
5. 회계(accounting) 정보: 이 정보는 CPU 사용 시간과 경과된 실시간, 시간 제한, 계정 번호, 잡 또는 프로세스 번호 등을 포함한다
6. 입출력 상태 정보: 이 정보는 이 프로세스에 할당된 입출력 장치들과 열린 파일의 목록 등을 포함한다.

프로세스 상태
프로세스 번호
프로그램 카운터
레지스터
메모리 제한
오픈 파일 리스트
. . . . .

3.1.4. Threads

프로세스가 단일의 실행 스레드를 실행하는 프로그램입니다.사용자는 문자를 입력하면서 동시에 철자 검사기를 실행할 수 없다. 대부분의 현대 운영체제는 프로세스 개념을 확장하여 한프로세스가 다수의 실행 스레드를 가질 수 있도록 허용한다. 그들은 따라서 프로세스가한 번에 하나 이상의 일을 수행할 수 있도록 허용한다. 이러한 특성은 특히 다중 처리기시스템에서 이익을 얻을 수 있는데, 여러 스레드가 병렬로 실행될 수 있다.

3.2. Process Scheduling

다중 프로그래밍의 목적은 CPU 이용을 최대화하기 위하여 항상 어떤 프로세스가 실행되도록 하는 데 있다. 단일 CPU 코어가 있는 시스템의 경우 한 번에 2개 이상의 프로세스가 실행될 수 없지만 다중 코어 시스템은 한 번에 여러 프로세스를 실행할 수 있다. 코어보다 많은 프로세스가 있는 경우 초과 프로세스는 코어가 사용 가능해지고 다시 스케줄 될 때까지 기다려야 한다. 현재 메모리에 있는 프로세스 수를 다중 프로그래밍 정도라고 한다.

3.2.1. Scheduling Queues

프로세스가 시스템에 들어가면 준비 큐에 들어가서 준비 상태가 되어 CPU 코어에서 실행되기를 기다린다. 이 큐는 일반적으로 연결 리스트로 저장된다. 준비 큐 헤더에는 리스트의 첫 번째 PCB에 대한 포인터가 저장되고 각 PCB에는 준비 큐의 다음 PCB를 가리키는 포인터 필드가 포함된다.

시스템에는 다른 큐도 존재한다. 프로세스에 CPU 코어가 할당되면 프로세스는 잠시 동안 실행되어 결국 종료되거나 인터럽트 되거나 I/O 요청의 완료와 같은 특정 이벤트가 발생할 때까지 기다린다. 프로세스가 디스크와 같은 장치에 I/O 요청을 한다고 가정하자. 장치는 프로세서보다 상당히 느리게 실행되므로 프로세스는 I/O가 사용 가능할 때까지 기다려야 한다. I/O 완료와 같은 특정 이벤트가 발생하기를 기다리는 프로세스는 대기 큐(wait queue)에 삽입된다

3.2.2. CPU Scheduling

프로세스는 수명주기 동안 준비 큐와 다양한 대기 큐를 이주한다. CPU 스케줄러의 역.할은 준비 큐에 있는 프로세스 중에서 선택된 하나의 프로세스에 CPU 코어를 할당하는 것이다. CPU 스케줄러는 CPU를 할당하기 위한 새 프로세스를 자주 선택해야 한다.

일부 운영체제는 스와핑으로 알려진 중간 형태의 스케줄링을 가지고 있는데, 핵심아이디어는 때로는 메모리에서(및 CPU에 대한 능동적 경쟁에서) 프로세스를 제거하여다중 프로그래밍의 정도를 감소시키는 것이 유리할 수 있다는 것이다. 나중에 프로세스를 메모리에 다시 적재될 수 있으며 중단된 위치에서 실행을 계속할 수 있다. 프로세스를 메모리에서 디스크로 "스왑아웃"하고 현재 상태를 저장하고, 이후 디스크에서 메모리로 "스왑인"하여 상태를 복원할 수 있기 때문에 이 기법을 스와핑이라고 한다. 스와핑은일반적으로 메모리가 초과 사용되어 가용공간을 확보해야 할 때만 필요하다.

3.2.3. Context Switch

인터럽트는 운영체제가 CPU 코어를 현재 작업에서 벗어 내어 커널 루틴을 실행할 수 있게 한다. 이러한 연산은 범용 시스템에서는 자주 발생한다.인터럽트가 발생하면 시스템은 인터럽트 처리가 끝난 후에 문맥을 복구할 수 있도록 현재 실행 중인 프로세스의 현재 문맥을 저장할 필요가 있다. 이는 결국 프로세스를 중단했다가 재개하는 작업이다. 문맥은 프로세스의 PCB에 표현된다. 문맥은 CPU 레지스터의 값, 프로세스 상태, 메모리 관리 정보 등을 포함한다. 일반적으로 커널모드이건 사용자 모드이건 CPU의 현재 상태를 저장하는 작업을 수행하고(state save),나중에 연산을 재개하기 위하여 상태 복구 작업을 수행한다(state restore).

CPU 코어를 다른 프로세스로 교환하려면 이전의 프로세스의 상태를 보관하고 새로운 프로세스의 보관된 상태를 복구하는 작업이 필요하다. 이 작업은 문맥 교환(context switch)이라고 한다.

문맥 교환 시간은 하드웨어의 지원에 크게 좌우된다. 또한운영체제가 복잡할수록, 문맥 교환 시 해야 할 작업의 양이 더 많아진다.

3.3. Operations on Processes

대부분 시스템 내의 프로세스들은 병행 실행될 수 있으며, 반드시 동적으로 생성되고, 제거되어야 한다. 그러므로 운영체제는 프로세스 생성 및 종료를 위한 기법을 제공해야 한다. 이 절에서는 프로세스 생성 기법에 대해 살펴보고 UNIX와 Window 시스템에서의프로세스 생성에 관해 설명한다.

3.3.1. Process Creation

실행되는 동안 프로세스는 여러 개의 새로운 프로세스들을 생성할 수 있다. 앞에서 언급한 것과 같이 생성하는 프로세스를 부모 프로세스라고 부르고, 새로운 프로세스는 자식프로세스라고 부른다. 이 새로운 프로세스들은 각각 다시 다른 프로세스들을 생성할 수있으며, 그 결과 프로세스의 트리를 형성한다.

UNIX, Linux 및 Windows와 같은 대부분의 현대 운영체제들은 유일한 프로세스식별자(pid)를 사용하여 프로세스를 구분하는데 이 식별자는 보통 정수이다. pid는 시스템의 각 프로세스에 고유한 값을 가지도록 할당된다. 이 식별자를 통하여 커널이 유지하고 있는 프로세스의 다양한 속성에 접근하기 위한 찾아보기(index)로 사용된다.

프로세스가 새로운 프로세스를 생성할 때, 두 프로세스를 실행시키는 데 두 가지 가능한 방법이 존재한다.

1. 부모는 자식과 병행하게 실행을 계속한다.
2. 부모는 일부 또는 모든 자식이 실행을 종료할 때까지 기다린다.

새로운 프로세스들의 주소 공간 측면에서 볼 때 다음과 같은 두 가지 가능성이 있다.

1. 자식 프로세스는 부모 프로세스의 복사본이다 (자식 프로세스는 부모와 똑같은 프로그램과 데이터를 가진다).
2. 자식 프로세스가 자신에게 적재될 새로운 프로그램을 가지고 있다.

3.3.2. Process Termination

프로세스가 마지막 문장의 실행을 끝내고, exit 시스템 콜을 사용하여 운영체제에 자신의 삭제를 요청하면 종료한다. 이 시점에서, 프로세스는 자신을 기다리고 있는 부모 프로세스에(wait 시스템 콜을 통해) 상태 값(통상 정수값)을 반환할 수 있다. 물리 메모리와가상 메모리, 열린 파일, 입출력 버퍼를 포함한 프로세스의 모든 자원이 할당 해제되고운영체제로 반납된다.

프로세스 종료가 발생하는 다른 경우가 있다. 한 프로세스는 적당한 시스템 콜 (예를들어, Windows의 TerminateProcess ())을 통해서 다른 프로세스의 종료를 유발할수 있다. 통상적으로, 그런 시스템 콜은 단지 종료될 프로세스의 부모만이 호출할 수 있다. 그렇지 않으면, 사용자 또는 오작동하는 프로세스가 다른 사용자의 프로세스를 임의로 중단(kill)시킬 수 있을 것이다. 부모가 자식을 종료시키기 위해서는 자식의 pid를 알아야 한다는 것에 유의하자. 그러므로 한 프로세스가 새로운 프로세스를 만들 때, 새로만들어진 프로세스의 신원(identity)이 부모에게 전달된다.

부모는 다음과 같이 여러 가지 이유로 인하여 자식 중 하나의 실행을 종료할 수 있다.

1. 자식이 자신에게 할당된 자원을 초과하여 사용할 때. 이때는 부모가 자식들의 상태를 검사할 수 있는 방편이 주어져야 한다.
2. 자식에게 할당된 태스크가 더 이상 필요 없을 때

3. 부모가 exit를 하는데, 운영체제는 부모가 exit 한 후에 자식이 실행을 계속하는 것을 허용하지 않는 경우

몇몇 시스템에서는 부모 프로세스가 종료한 이후에 자식 프로세스가 존재할 수 없다. 그러한 시스템에서는 프로세스가 종료되면 (정상적이든 비정상적이든) 그로부터 비롯된 모든 자식 프로세스들도 종료되어야 한다. 이것을 연쇄식 종료(cascading termination)라고 부르며 이 작업은 운영체제가 시행한다

프로세스 실행과 종료를 설명하기 위해서 Linux와 UNIX 시스템에서 exit () 시스템 콜을 사용하여 프로세스를 종료시키는 것을 고려해 보자. exit () 시스템 콜은 종료 상태를 나타내는 인자를 전달받는다.

```
/* 1인 상태로 exit 1 */
exit(1);
```

사실 정상적인 종료에서 exit ()는 위와 같이 직접적으로 호출되거나 UNIX 실행 파일에 추가되는 C 실행시간 라이브러리가 디폴트로 exit () 호출을 추가하기 때문에 간접적으로 호출될 수 있다.

부모 프로세스는 wait () 시스템 콜을 사용해서 자식 프로세스가 종료할 때를 기다 릴 수 있다. wait () 시스템 콜은 부모가 자식의 종료 상태를 얻어낼 수 있도록 하나의인자를 전달받는다. 이 시스템 콜은 부모가 어느 자식이 종료되었는지 구별할 수 있도록종료된 자식의 프로세스 식별자를 반환한다.

```
pid_t pid;
int status;
pid = wait (&status);
```

프로세스가 종료하면 사용하던 자원은 운영체제가 되찾아 간다. 그러나 프로세스의종료 상태가 저장되는 프로세스 테이블의 해당 항목은 부모 프로세스가 wait ()를 호출할 때까지 남아 있게 된다. 종료되었지만 부모 프로세스가 아직 wait () 호출을 하지 않은 프로세스를 좀비(zombie) 프로세스라고 한다. 종료하게 되면 모든 프로세스는 좀비상태가 되지만 아주 짧은 시간 동안만 머무른다. 부모가 wait ()를 호출하면 좀비 프로세스의 프로세스 식별자와 프로세스 테이블의 해당 항목이 운영체제에 반환된다.

부모 프로세스가 wait ()를 호출하는 대신 종료한다면 무슨 일이 벌어지는 살펴보자. 이 상황에 부닥친 자식 프로세스를 고아(orphan) 프로세스라고 부른다. 전통적인 UNIX는 고아 프로세스의 새로운 부모 프로세스로 init 프로세스를 지정함으로써 이문제를 해결한다(3.3.1절에서 init 프로세스가 UNIX 시스템에서 프로세스 계층 구조의 루트에 위치한다는 사실을 상기하라.) init 프로세스는 주기적으로 wait ()를 호출하여 고아 프로세스의 종료 상태를 수집하고 프로세스 식별자와 프로세스 테이블 항목을 반환한다.

대부분의 Linux 시스템은 init를 systemd로 대체했지만, 후자의 프로세스는 여전히 동일한 역할을 수행할 수 있지만 Linux는 systemd 이외의 프로세스가 고아 프로세스를 상속하고 종료를 관리하도록 허용한다.

3.3.2.1. Android Process Hierarchy

제한된 메모리와 같은 자원 제약 때문에 모바일 운영체제는 제한된 시스템 자원을 회수하기 위해 기존 프로세스를 종료해야 할 수도 있다. Android는 임의의 프로세스를 종료하지 않고 프로세스의 중요도 계층을 식별했으며, 시스템이 프로세스를 종료하여 새로운 또는 보다 중요한 프로세스를 위한 자원을 확보해야 할 경우 중요도가 낮은 프로세스부터 종료한다. 가장 중요한 순서부터 프로세스를 분류하면 다음과 같다.

- 전경 프로세스(foreground process) -사용자가 현재 상호 작용하고 있는 응용 프로그램을 나타내며, 화면에 보이는 현재 프로세스
- 가시적 프로세스(visible process) -전경에서 직접 볼 수 없지만 전경 프로세스가 참조하는 활동 (즉, 현재 상태가 전경 프로세스에 표시되는 활동을 수행하는 프로세스)을 수행하는 프로세스
- 서비스 프로세스(service process) -백그라운드 프로세스와 유사하지만 사용자가 인지할 수 있는 활동(예: 음악 스트리밍)을 수행하는 프로세스
- 백그라운드 프로세스(background process) -활동을 수행하고 있지만 사용자가 인식하지 못하는 프로세스
- 빈 프로세스(empty process)-응용 프로그램과 관련된 활성 구성요소가 없는 프로세스

3.4. Interprocess Communication

운영체제 내에서 실행되는 병행 프로세스들은 독립적이거나 또는 협력적인 프로세스들일 수 있다. 프로세스가 시스템에서 실행 중인 다른 프로세스들과 데이터를 공유하지 않는 프로세스는 독립적이다. 프로세스가 시스템에서 실행 중인 다른 프로세스들에 영향을주거나 받는다면 이는 협력적인 프로세스들이다. 분명히 다른 프로세스들과 자료를 공유하는 프로세스는 상호 협력적인 프로세스이다.

1. 정보 공유(information sharing)
2. 계산 가속화(computation speedup)
3. 모듈성(modularity)

협력적 프로세스들은 데이터를 교환할 수 있는, 즉 서로 데이터를 보내거나 받을 수있는 프로세스 간 통신(interprocess communication, IPC) 기법이 필요하다. 프로세스 간 통신에는 기본적으로 공유 메모리(shared memory)와 메시지 전달(message passing)의 두 가지 모델이 있다.

3.4.1. Shared-Memory Systems

공유 메모리를 사용하는 프로세스 간 통신에서는 통신하는 프로세스들이 공유 메모리영역을 구축해야 한다. 통상 공유 메모리 영역은 공유 메모리 세그먼트를 생성하는 프로세스의 주소 공간에 위치한다. 이 공유 메모리 세그먼트를 이용하여 통신하고자 하는 다른 프로세스들은 이 세그먼트를 자신의 주소 공간에 추가하여야 한다.

협력하는 프로세스의 개념을 설명하기 위해서, 협력하는 프로세스의 일반적인 패러다임인 생산자-소비자 문제를 생각해 보기로 하자. 생산자 프로세스는 정보를 생산하고소비자 프로세스는 정보를 소비한다. 예를 들어, 컴파일러는 어셈블리 코드를 생산하고,어셈블러는 이것을 소비한다. 어셈블러는 이어 목적 모듈(object module)을 생산할 수있고, 적재기(loader)는 이들을 소비한다. 생산자 소비자 문제는 클라이언트 서버 패러다임을 위한 유용한 은유를 제공한다. 일반적으로 우리는 서버를 생산자로 클라이언트를소비자로 생각한다. 예를 들면 웹 서버는 HTML 파일과 이미지와 같은 웹 콘텐츠를 생산하고(즉, 제공하고) 이 자원들을 요청한 클라이언트 웹 브라우저가 소비하게 된다(즉,읽는다).

생산자-소비자 문제의 하나의 해결책은 공유 메모리를 사용하는 것이다. 생산자와소비자 프로세스들이 병행으로 실행되도록 하려면, 생산자가 정보를 채워 넣고 소비자가소모할 수 있는 항목들의 버퍼가 반드시 사용 가능해야 한다. 이 버퍼는 생산자와 소비자가 공유하는 메모리 영역에 존재하게 된다. 생산자가 한 항목을 생산하고, 그동안에 소비자는 다른 항목을 소비할 수 있다. 생산자와 소비자가 반드시 동기화되어야 생산되지도 않은 항목들을 소비자가 소비하려고 시도하지 않을 것이다.

두 가지 유형의 버퍼가 사용된다. 무한 버퍼(unbounded buffer)의 생산자 소비자 문제에서는 버퍼의 크기에 실질적인 한계가 없다. 소비자는 새로운 항목을 기다려야만 할 수도 있지만, 생산자는 항상 새로운 항목을 생산할 수 있다. 유한 버퍼(boundedbuffer)는 버퍼의 크기가 고정되어 있다고 가정한다. 이 경우 버퍼가 비어 있으면 소비자는 반드시 대기해야 하며, 모든 버퍼가 채워져 있으면 생산자가 대기해야 한다.

유한 버퍼가 공유 메모리를 사용한 프로세스간 통신을 어떻게 분명하게 하는지 살펴보자. 다음 변수들은 생산자와 소비자 프로세스가 공유하는 메모리 영역에 존재한다.

```
#define BUFFER_SIZE 10
typedef struct {
        ...
} item;
item buffer [BUFFER_SIZE] ;
int in = 0;
int out = 0;
```

공유 버퍼는 두 개의 논리 포인터 in과 out을 갖는 원형 배열로 구현된다. 변수 in은 버퍼 내에서 다음으로 비어 있는 위치를 가리키며, out은 버퍼 내에서 첫 번째로 채워져 있는 위치를

가리킨다. in == out; 일 때 버퍼는 비어 있고, ((in+ 1)% BUFFER_SIZE) == out)이면 버퍼는 가득 차 있다.

생산자와 소비자의 코드가 그림 3.12와 3.13에 각각 나와 있다. 생산자 프로세스는next_Produced라는 지역 변수에 다음번 생산되는 item을 저장하고 있다. 소비자 코드는 next_Consumed라는 지역 변수에 다음번 소비되는 item을 저장하고 있다.

```
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;
}
```

[그림 3.12]

```
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 */
}
```

[그림 3.13]

3.4.2. Message-Passing Systems

서로 협력하는 프로세스가 공유 메모리 환경에서 어떻게 상호 통신을 행하는지 알아보았다. 그 기법은 프로세스들이 메모리 영역을 공유할 것을 필요로 하며, 공유메모리에 접근하고 조작하는 코드가 응용 프로그래머에 의해 명시적으로 작성되어야 했다.

메시지 전달 방식은 동일한 주소 공간을 공유하지 않고도 프로세스들이 통신을 하고, 그들의 동작을 동기화할 수 있도록 허용하는 기법을 제공한다. 메시지 전달 방식은통신하는 프로세스들이 네트워크에 의해 연결된 다른 컴퓨터들에 존재할 수 있는 분산환경에서 특히 유용하다. 한 예로 월드 와이드 웹에 사용되는 chat 프로그램은 서로 메시지를 교환하여 통신하도록 설계될 수 있다.

메시지 전달 시스템은 최소한 두 가지 연산을 제공한다.

- send (message)
- receive (message)

만약 프로세스 P와 Q가 통신을 원하면, 반드시 서로 메시지를 보내고 받아야 한다.이들 사이에 통신 연결(communication link)이 설정되어야 한다. 이 연결은 다양한 방법으로 구현할 수 있다. 우리는 연결의 물리적인 구현에 관심이 있는 것이 아니라, 논리적인 구현에 관심이 있다. 하나의 링크와 send () / receive() 연산을 논리적으로 구현하는 다수의 방법은 다음과 같다.

- 직접 또는 간접 통신
- 동기식 또는 비동기식 통신
- 자동 또는 명시적 버퍼링

3.4.2.1. Naming

통신을 원하는 프로세스들은 서로를 가리킬 방법이 있어야 한다. 이들은 간접 통신 또는직접 통신을 사용할 수 있다.

직접 통신하에서, 통신을 원하는 각 프로세스는 통신의 수신자 또는 송신자의 이름을 명시해야 한다. 이 기법에서 send (), receive () 프리미티브들은 다음과 같이 정의한다.

- send (P, message)-프로세스 P에 메시지를 전송한다.
- receive (Q, message)-프로세스 Q로부터 메시지를 수신한다.

이 기법에서 통신 연결은 다음의 특성을 가진다.

- 통신을 원하는 각 프로세스의 쌍들 사이에 연결이 자동으로 구축된다.
  프로세스들은통신하기 위해 상대방의 신원(identity)만 알면 된다.
- 연결은 정확히 두 프로세스 사이에만 연관된다.
- 통신하는 프로세스들의 각 쌍 사이에는 정확하게 하나의 연결이 존재해야 한다.

이 기법은 주소 방식에서 대칭성을 보인다. 즉, 송신자와 수신자 프로세스가 모두 통신하려면 상대방의 이름을 제시해야 한다. 이 기법의 변형으로서 주소 지정 시에 비대칭을 사용할 수도 있다. 송신자만 수신자 이름을 지명하며, 수신자는 송신자의 이름을 제시할 필요가 없다. 이 기법에서 send ()와 receive () 프리미티브들을 다음과 같이 정의한다.

- send (P, message)-메시지를 프로세스 P에 전송한다.
- receive (id, message)-임의의 프로세스로부터 메시지를 수신한다. 변수 id는통신을 발생시킨 프로세스의 이름으로 설정된다.

간접 통신에서 메시지들은 메일박스(mailbox) 또는 포트(port)로 송신되고, 그것으로부터 수신된다. 메일박스는 추상적으로 프로세스들에 의해 메시지들이 넣어지고, 메시지들이 제거될 수 있는 객체라고도 볼 수 있다. 각 메일박스는 고유의 id를 가진다. 예를 들어 POSIX 메시지 큐는 메일박스를 식별하기 위하여 정수 값을 사용한다. 이 기법에서 프로세스는 다수의 상이한 메일박스를 통해 다른 프로세스들과 통신할 수 있다. 두프로세스들이 공유 메일박스를 가질 때만 이들 프로세스가 통신할 수 있다. send ()와receive () 프리미티브들은 다음과 같이 정의할 수 있다.

- send (A, message)-메시지를 메일박스 A로 송신한다.
- receive (A, message)-메시지를 메일박스 A로부터 수신한다.

이 방법에서 통신 연결은 다음의 성질을 가진다

- 한 쌍의 프로세스들 사이의 연결은 이들 프로세스가 공유 메일박스를 가질 때만 구축된다.
- 연결은 두 개 이상의 프로세스들과 연관될 수 있다.
- 통신하고 있는 각 프로세스 사이에는 다수의 서로 다른 연결이 존재할 수 있고, 각연결은 하나의 메일박스에 대응된다.

운영체제가 소유한 메일박스는 자체적으로 존재한다. 이것은 독립적인 것으로 어떤 특정한 프로세스에 예속되지 않는다. 운영체제는 한 프로세스에 다음을 할 수있도록 허용하는 기법을 반드시 제공해야 한다.

- 새로운 메일박스를 생성한다.
- 메일박스를 통해 메시지를 송신하고 수신한다.
- 메일박스를 삭제한다.

3.4.2.2. Synchronization

프로세스 간의 통신은 send와 receive 프리미티브에 대한 호출에 의해 발생한다. 각프리미티브를 구현하기 위한 서로 다른 설계 옵션이 있다. 메시지 전달은 봉쇄형 (blocking)이거나 비봉쇄형 (nonblocking) 방식으로 전달된다. 이 두 방식은 각각 동기식, 비동기식이라고도 알려져 있다

1. 봉쇄형 보내기: 송신하는 프로세스는 메시지가 수신 프로세스 또는 메일박스에 의해수신될 때까지 봉쇄된다.
2. 비봉쇄형 보내기: 송신하는 프로세스가 메시지를 보내고 작업을 재시작한다.
3. 봉쇄형 받기: 메시지가 이용 가능할 때까지 수신 프로세스가 봉쇄된다.
4. 비봉쇄형 받기: 송신하는 프로세스가 유효한 메시지 또는 널(null)을 받는다.

3.4.2.3. Buffering

통신이 직접적이든 간접적이든 간에, 통신하는 프로세스들에 의해 교환되는 메시지는 임시 큐에 들어 있다. 기본적으로 이러한 큐를 구현하는 방식은 세 가지가 있다.

1. 무용량(zero capacity) : 큐의 최대 길이가 0이다. 즉, 링크는 자체 안에 대기하는 메시지들을 가질 수 없다. 이 경우에, 송신자는 수신자가 메시지를 수신할 때까지 기다려야 한다.
2. 유한 용량(bounded capacity): 큐는 유한한 길이 n을 가진다. 즉, 최대 n개의 메시지가 그 안에 들어 있을 수 있다. 새로운 메시지가 전송될 때 큐가 만원이 아니라면,메시지는 큐에 놓이며(메시지가 복사되든지 또는 메시지에 대한 포인터가 유지된다), 송신자는 대기하지 않고 실행을 계속한다. 그렇지만, 링크는 유한한 용량을 가진다. 링크가 만원이면, 송신자는 큐 안에 공간이 이용 가능할 때까지 반드시 봉쇄되어야 한다.
3. 무한 용량(unbounded capacity): 큐는 잠재적으로 무한한 길이를 가진다. 따라서메시지들이 얼마든지 큐 안에서 대기할 수 있다. 송신자는 절대 봉쇄되지 않는다.

3.5. Examples of IPC Systems

3.5.1. An Example: POSIX Shared Memory

공유 메모리와 메시지 전달을 포함하여 POSIX 시스템을 위한 다수의 IPC 기법이 사용가능하다. 여기서 공유 메모리를 위한 POSIX API를 살펴본다.

POSIX 공유 메모리는 메모리-사상 파일을 사용하여 구현된다. 메모리-사상 파일은 공유 메모리의 특정 영역을 파일과 연관시킨다. 프로세스는 먼저 아래와 같이 shm_open () 시스템 콜을 사용하여 공유 메모리 객체를 생성해야 한다.

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

첫 번째 인자는 공유 메모리 객체의 이름을 지정한다. 공유 메모리에 접근하고자 하는프로세스는 이 이름을 통하여 객체를 언급한다. 이어서 두 번째 인자는 객체가 존재하지않으면 생성되고(O_CREAT) 객체는 읽기와 쓰기가 가능한 상태로 열린다는 것을 나타낸다(O_RDRW). 마지막 인자는 공유 메모리 객체에 파일-접근 허가권을 부여한다. shm_open()이 성공하면 공유 메모리 객체를 나타내는 정수형 파일 설명자를 반환한다.

객체가 설정되면 ftruncate () 함수를 사용하여 객체의 크기를 바이트 단위로 설정한다. 아래와 같은 호출은 그림 3.16과 3.17의 프로그램은 공유 메모리를 구현하기 위하여 생산자-소비자 모델을 사용한다. 생산자는 공유 메모리 객체를 구축하고 공유 메모리에 데이터를 쓰고, 소비자는 공유 메모리에서 데이터를 읽는다.

```
ftruncate (fd, 4096);
```

그림 3.16에 보인 생산자는 OS라고 명명된 공유 메모리 객체를 생성하고 악명 높은 문자열 "Hello World!"를 공유 메모리에 쓴다. 프로그램은 지정된 크기의 공유 메모리 객체를 메모리에 사상하고 객체에 쓰기 권한을 부여한다. MAP_SHARED 플래그는공유 메모리 객체에 변경이 발생하면 객체를 공유하는 모든 프로세스가 최신의 값을 접근하게 된다는 것을 지정한다. 공유 메모리 객체에 쓰기 작업을 할 때 sprintf() 함수를 호출하고 출력 형식이 완성된 문자열은 ptr이 가리키는 공유 메모리 객체에 쓰인다는 것을 주의하라. 쓰기 작업이 성공하면 쓰인 바이트 수만큼 포인터를 반드시 증가시켜야 한다.

그림 3.17의 소비자 프로세스는 공유 메모리의 내용을 읽고 출력한다. 또한 소비자는 shm_unlink() 함수를 호출하여 접근이 끝난 공유 메모리를 제거한다. 3장 마지막의 프로그래밍 연습문제에서 POSIX 공유 메모리 API를 사용하는 추가의 연습문제를제공한다. 덧붙여 13.5절에서 메모리 사상에 대해 더 상세한 설명을 한다.

```
message next_consumed;

while (true) {
     receive(next_consumed);

     /* consume the item in next_consumed */
}
```

[그림 3.16]

```
#include <stdio.h>
#include <stlib.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_RDRW, 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;
}
```

[그림 3.17]

3.5.2. An Example: Mach

다음 메시지 전달의 예로 Mach 운영체제를 고려한다. Mach는 특히 분산 시스템용으로 설계되었지만 2장에서 논의된 것처럼 macOS 및 iOS 운영체제에 포함된 사실이 입증하듯이 데스크톱 및 모바일 시스템에도 적합하다.

Mach 커널은 프로세스와 유사하지만 제어 스레드가 많고 관련 자원이 적은 다중태스크의 생성 및 제거를 지원한다. 모든 태스크 간 통신을 포함하여 Mach에서 대부분의 통신은 메시지로 수행된다. Mach에서 포트(port)라고 하는 메일박스로 메시지를 주고 받는다. 포트는 크기가 정해져 있고 단방향이다. 양방향 통신의 경우 메시지가 한 포트로 전송되고 응답이 별도의 응답 포트로 전송된다. 각 포트에는 여러 송신자가 있을수 있지만 수신자는 오직 하나만 존재한다. Mach는 포트를 사용하여 태스크, 스레드,메모리 및 프로세서와 같은 자원을 나타내며, 메시지 전달은 이러한 시스템 자원 및 서비스와 상호 작용하기 위한 객체 지향 접근 방식을 제공한다.

동일한 호스트 또는 분산시스템의 별도 호스트의 두 포트 사이에서 메시지 전달이 발생할 수 있다.

각 포트에는 그 포트와 상호 작용하는 데 필요한 자격을 식별하는 포트 권한 집합이연관된다. 예를 들어, 태스크가 포트에서 메시지를 수신하려면 해당 포트에 대해 MACH_PORT_RIGHT_RECEIVE 자격이 있어야 한다. 포트를 생성한 태스크가 해당 포트의 소유자이며, 소유자는 해당 포트에서 메시지를 수신할 수 있는 유일한 태스크이다. 포트의 소유자는 포트의 자격을 조작할 수도 있다. 이러한 조작은 일반적으로 응답 포트를 설정할때 수행된다. 예를 들어, 태스크 T1이 포트 P1을 소유하고 태스크 P2가 소유한 포트 P2에 메시지를 전송한다고 가정하자. T1이 T2로부터 응답을 받으려면 T2에 포트 P1에 대한MACH_PORT_RIGHT_SEND 권한을 부여해야 한다. 포트 권한의 소유권은 태스크에게주어진다. 즉, 동일한 태스크에 속하는 모든 스레드가 동일한 포트 권한을 공유한다. 따라서 동일한 태스크에 속하는 두 개의 스레드는 각 스레드와 관련된 스레드-별 포트를통해 메시지를 교환하여 쉽게 통신할 수 있다.

태스크가 생성되면 Task Self 포트와 Notifu 포트라는 두 개의 특별한 포트도 생성된다. 커널은 Task Self 포트에 대한 수신 권한을 가지고 있어 태스크가 커널에 메시지를 보낼 수 있다. 커널은 이벤트 발생 알림을 작업의 Notify 포트(물론 태스크가 수신권한을 가지고 있음)로 보낼 수 있다.

mach_port_allocate () 함수 호출은 새 포트를 작성하고 메시지 큐를 위한 공간을 할당한다. 또한 포트에 대한 권한을 식별한다. 각 포트 권한은 해당 포트의 이름을나타내며 포트는 권한을 통해서만 액세스 할 수 있다. 포트 이름은 단순한 정수 값이며UNIX 파일 디스크립터와 매우 유사하게 작동한다. 다음 예제는 이 API를 사용하여 포트를 생성하는 방법을 보여준다.

```
mach port.t port; // 포트 권한의 이름

mach port allocate(
   mach task self (), // a task referring to itself
   MACH PORT RIGHT RECEIVE, // the right for this port
   &port); // the name of the port right
```

각 태스크는 또한 부트스트랩 포트에 액세스 할 수 있어서 태스크가 생성한 포트를 시스템 전체의 부트스트랩 서버에 등록할 수 있다. 포트가 부트스트랩 서버에 등록되면다른 태스크가 이 레지스트리에서 포트를 검색하여 포트로 메시지를 보낼 수 있는 권한을 얻을 수 있다.

각 포트와 관련된 큐는 크기가 제한되어 있으며 처음에는 비어 있다. 메시지가 포트로 전송되면 큐에 복사된다. 모든 메시지는 안정적으로 전달되며 동일한 우선순위를 가진다. Mach는 동일한 송신자의 여러 메시지가 선입선출(FIFO) 순서로 큐에 삽입 하지만 절대적 순서를 보장하지는 않는다. 예를 들어, 두 명의 송신자가 보낸 메시지는 임의의 순서로 큐에 저장된다.

Mach 메시지에는 다음 두 필드를 포함한다.

1. 고정 크기의 메시지 헤더. 헤더는 메시지 크기, 소스 및 대상 포트를 포함한 메시지에 관한 메타 데이터를 포함한다. 일반적으로 송신 스레드는 응답을 예상하므로 소스의 포트 이름이 수신 태스크로 전달되어 응답을 보내는 데 "반환 주소"로 사용할수 있다.
2. 데이터를 포함하는 가변 크기 본체

메시지는 단순하거나 복잡할 수 있다. 간단한 메시지는 커널에 의해 해석되지 않는구조화되지 않은 보통의 사용자 데이터를 포함한다. 복잡한 메시지는 "out-of-line" 데이터를 포함하는 메모리 위치에 대한 포인터를 포함하거나 다른 태스크에 포트 권한을전송하는 데 사용될 수 있다. Out-of-line 데이터 포인터는 메시지가 많은 양의 데이터를 전달해야 할 때 특히 유용하다. 간단한 메시지는 메시지의 데이터를 복사하고 패키징해야 한다. Out-of-line 데이터 전송에는 데이터가 저장된 메모리 위치를 가리키는 포인터만 필요하다.

mach_msg () 함수는 메시지를 보내고 받는 표준 API이다. 함수의 매개변수 중 하나가 MACH_SEND_MSG 또는 MACH_RCV_MSG 값을 가지며 송신 또는 수신 연산인지를나타낸다. 이제 클라이언트 태스크가 서버 태스크에 간단한 메시지를 보낼 때 사용되는 방법을 설명한다. 클라이언트 및 서버 태스크와 각각 연관된 두 개의 포트(client와server)가 있다고 가정하자. 그림 3.18의 코드는 클라이언트 태스크가 헤더를 구성하고 서버로 메시지를 보내는 것뿐만 아니라 클라이언트가 보낸 메시지를 받는 서버 태스크를 보여준다.

mach_msg () 함수 호출은 메시지 전달을 수행하기 위해 사용자 프로그램에 의해 호출된다. 그런 다음 mach_msg ()는 mah_msg_trap () 함수를 호출한다. 이는 mah 커널에 대한 시스템 콜이다. 커널 내에서 mach_msg_trap ()은 mach_msg_overwrite_trap() 함수를 호출하여 메시지의 실제 전달을 처리한다.

송수신 작업 자체는 융통성이 있다. 예를 들어, 메시지가 포트로 전송되었을 때, 큐가 가득 찾을 수 있다. 큐가 가득 차지 않으면 메시지가 큐에 복사되고 전송 작업이 계속된다. 포트의 큐가 가득 찬 경우 송신자는 mach_msg ()의 매개변수를 통해 다음 중 하나를 선택할 수 있다.

```c
#include <stdio.h>
#include <stlib.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;
}
```

[그림 3.18]

3.5.3. An Example: Windows

Windows의 메시지 전달 설비는 고급 로컬 프로시저 호출 설비(advanced local procedure call facility, ALPC)라 불린다. ALPC는 동일 기계상에 있는 두 프로세스간의 통신에 사용한다. 이것은 널리 사용되는 표준 원격 프로시저 호출(RPC) 기법과 같으나, Windows에 맞게 특별히 최적화되었다(원격 프로시저 호출에 관해서는 3.8.2절에서 상세하게 다룬다). Mach와 유사하게, Windows는 두 프로세스 간에 연결을 구축하고 유지하기 위해 포트 객체를 사용한다. Windows는 연결 포트(connection port)와 통신 포트(communication port)의 두 가지 유형의 포트를 사용한다.

서버 프로세스는 모든 프로세스가 접근할 수 있는 연결 포트 객체를 공표한다. 클라이언트가 서브시스템으로부터 서비스를 원할 경우, 서버의 연결 포트 객체에 대한 핸들을 열고 연결 요청을 보낸다. 그러면 서버는 채널을 생성하고 핸들을 클라이언트에게 반환한다. 채널은 한 쌍의 사적인 통신 포트로 구성되는데, 하나는 클라이언트에서 서버로메시지를 보내기 위한 포트이고 다른 하나는 서버에서 클라이언트로 메시지를 보내기위한 포트이다. 추가적으로 통신 채널은 클라이언트와 서버가 응답 메시지를 기다리고있는 동안에도 다른 요청을 받아들일 수 있도록 콜백 기법을 제공한다.

ALPC 채널이 생성되면 다음 3가지 중 하나의 메시지 전달 기법의 하나가 선택된다.

1. 256바이트까지의 작은 메시지의 경우, 포트의 메시지 큐가 중간 저장소로 사용되고, 메시지는 프로세스에서 프로세스로 복사된다.
2. 대용량 메시지는 반드시 섹션 객체(section object)를 통하여 전달되어야 한다. 섹션 객체란 채널과 연관된 공유 메모리의 영역을 말한다.

3. 데이터의 양이 너무 많아서 섹션 객체에 저장될 수 없는 경우, 서버 프로세스가 클라이언트의 주소 공간을 직접 읽거나 쓸 수 있는 API를 사용할 수 있다.

3.6. Communication in Client–Server Systems

3.6.1. Sockets

소켓(socket)은 통신의 극점(endpoint)을 뜻한다. 두 프로세스가 네트워크상에서 통신을 하려면 양 프로세스마다 하나씩, 총 두 개의 소켓이 필요해진다. 각 소켓은 IP 주소와 포트 번호 두 가지를 접합(concatenate)해서 구별한다. 일반적으로 소켓은 클라이언트-서버 구조를 사용한다. 서버는 지정된 포트에 클라이언트 요청 메시지가 도착하기를기다리게 된다. 요청이 수신되면 서버는 클라이언트 소켓으로부터 연결 요청을 수락함으로써 연결이 완성된다. Telnet, ftp 및 http 등의 특정 서비스를 구현하는 서버는 wellknown 포트로부터 메시지를 기다린다("well-known"이란 전 세계적으로 표준으로 사용하는 포트 번호라는 의미). 예를 들면 SSH 서버는 22번, FTP 서버는 21번, HTTP 서버는 80번 포트를 사용한다. 1024 미만의 모든 포트는 well-known 포트로 간주되며 표준 서비스를 구현하는 데 사용된다.

Host X(146.86.5.20)	Web server (161.25.19.8)
소켓(146.86.5.20:1625)	소켓(161.25.19.8:80)

클라이언트 프로세스가 연결을 요청하면 호스트 컴퓨터가 포트 번호를 부여한다. 이번호는 1024보다 큰 임의의 정수가 된다. 예를 들면 IP 주소 146.86.5.20인 호스트 X에있는 클라이언트가 IP 주소 161.25.19.8의 웹 서버에 접속하려고 한다면 호스트 X는 클라이언트에 포트 1625를 부여한다(웹 서버는 포트 80을 listen 하고 있다). 연결은 이 두개의 소켓 호스트 X의 (146.86.5.20:1625)와 웹 서버의 (161.25.19.8:80)으로 구성된다.

예제 프로그램은 연결 기반 TCP 소켓을 사용하는 date 서버를 설명한다. 클라이언트는 이 서버로부터 현재 날짜와 시간을 알아볼 수 있다. 1024보다 큰 사용되지 않는 임의의 번호를 가질 수 있지만 예제에서는 서버가 포트 6013을 listen 하고 있다. 연결이수신되면 서버는 클라이언트에게 현재 날짜와 시간을 보내준다.

Date 서버가 그림 3.27에 나와 있다. 서버는 포트 6013을 listen 한다는 것을 지정하는 ServerSocket를 생성한다. 그런 후에 accept () 메소드를 이용하여 listen 하게된다. 서버는 accept () 메소드에서 클라이언트가 연결을 요청할 때까지 봉쇄된다. 연결 요청이 들어오면 accept ()는 클라이언트와 통신하기 위해 사용할 수 있는 소켓을 반환한다.

서버가 소켓과 통신하는 자세한 방법은 아래와 같다. 서버는 먼저 PrintWriter 객체를 만들고 이 객체는 추후 클라이언트와 통신하는 데 사용된다. PrintWriter 객체는서버가 print ()나 println ()과 같은 루틴을 써서 소켓에 데이터를 쓸 수 있게 한다.

서버는 println () 메소드를 호출하여 클라이언트에게 날짜를 보낸다. 날짜를 소켓에쓰면 서버는 이 클라이언트와의 소켓을 닫고 다른 요청을 기다리게 된다.

클라이언트는 소켓을 생성하고 서버가 listen 하는 포트와 연결함으로써 서버와 통신을 시작한다. 그림 3.28에 이러한 클라이언트를 Java 프로그램으로 구현하였다. 클라이언트는 Socket을 생성하고 IP 주소 127.0.0.1에 있는 포트 6013의 서버와 연결해 주기를 요청한다. 연결되면 소켓은 일반적인 스트림 I/O 명령문을 사용하여 그 소켓으로부터 읽을 수 있다. 서버로부터 날짜를 받은 후 클라이언트는 소켓을 닫고 종료한다. IP주소 127.0.0.1은 루프백(loopback)을 나타내는 특별한 IP 주소이다. 컴퓨터가 IP 주소127.0.0.1을 참조하면 그 자신 기계를 지칭하는 것이다. 이처럼 하면 같은 기계에 있는클라이언트와 서버가 TCP/IP 프로토콜을 사용하여 통신하게 된다. IP 주소 127.0.0.1대신 date 서버를 실행하고 있는 원격지 호스트의 IP 주소를 사용할 수도 있다. IP 주소뿐 아니라 www.westminstercollege.edu와 같은 실제 호스트 이름을 사용할 수도있다.

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

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE ReadHandle, WriteHandle;
STARTUPINFO si;
PROCESS_INFORMATION pi;
char message[BUFFER_SIZE] = "Greetings";
DWORD written;
```

[그림 3.27 - 그림 3.28로 이어진다]

```
/* set up security attributes allowing pipes to be inherited */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
/* allocate memory */
ZeroMemory(&pi, sizeof(pi));

/* create the pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
   fprintf(stderr, "Create Pipe Failed");
   return 1;
}

/* establish the START_INFO structure for the child process */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* redirect standard input to the read end of the pipe */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* don't allow the child to inherit the write end of pipe */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* create the child process */
CreateProcess(NULL, "child.exe", NULL, NULL,
 TRUE, /* inherit handles */
 0, NULL, NULL, &si, &pi);

/* close the unused end of the pipe */
CloseHandle(ReadHandle);

/* the parent writes to the pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
   fprintf(stderr, "Error writing to pipe.");

/* close the write end of the pipe */
CloseHandle(WriteHandle);

/* wait for the child to exit */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

[그림 3.28 - 그림 3.27에서 이어진다]

3.6.2. Remote Procedure Calls, RPC

원격 서비스와 관련한 가장 보편적인 형태 중 하나는 RPC 패러다임으로서, 네트워크에연결된 두 시스템 사이의 통신에 사용하기 위하여 프로시저 호출 기법을 추상화하는 방법으로 설계되었다. 이것은 3.4절에서 소개한 IPC와 많은 측면에서 유사하며 사실 그러한 IPC 기반 위에 만들어진다. 그러나 여기서는 프로세스들이 서로 다른 시스템 위에서 돌아가기 때문에 원격 서비스를 제공하기 위해서는 메시지 기반 통신을 해야 한다.

IPC 방식과는 달리 RPC 통신에서 전달되는 메시지는 구조화되어 있고, 따라서 데이터의 패킨 수준을 넘어서게 된다. 각 메시지에는 원격지 포트에서 listen 중인 RPC디먼의 주소가 지정되어 있고 실행되어야 할 함수의 식별자, 그리고 그 함수에게 전달되어야 할 매개변수가 포함된다. 그런 후에 요청된 함수가 실행되고 어떤 출력이든지 별도의 메시지를 통해 요청자에게 반환된다.

이 문맥에서 포트는 단순히 메시지 패키지의 시작 부분에 포함되는 정수이다. 시스템은 일반적으로 네트워크 주소는 하나씩 가지지만 그 시스템에서 지원되는 여러 서비스를 구별하기 위해 포트를 여러 개 가질 수 있다. 원격 프로세스가 어떤 서비스를 받고자 하면 그 서비스에 대응되는 적절한 포트 주소로 메시지를 보내야 한다. 예를 들어한 시스템이 자신의 사용자

목록을 다른 시스템에서 알 수 있도록 원하면, 관련(그러한RPC 서비스를 제공해 주는) 디먼을 port 3027과 같은 곳에 등록시켜 놓는다. 그러면원격 시스템은 서버의 포트 3027로 RPC 메시지를 보내면 필요한 청보를(이 시스템의사용자 목록) 얻을 수 있다. 이 데이터는 응답 메시지 형태로 받게 된다.

RPC는 클라이언트가 원격 호스트의 프로시저 호출하는 것을 마지 차기의 프로시저 호출하는 것처럼 해준다. RPC 시스템은 클라이언트 쪽에 스럽을 제공하여 통신을 하는 데 필요한 자세한 사항들을 숨겨 준다. 보통 원격 프로시저마다 다른 스럽이 존재한다. 클라이언트가 원격 프로시저를 호출하면 RPC는 그에 대응하는 스럽을 호줄하고 원격 프로시저가 필요로 하는 매개변수를 건네준다. 그러면 스럽이 원격 서버의 포트를 찾고 매개변수를 정돈(marshall)한다. 그 후 스럽은 메시지 전달 기법을 사용하여 서버에게 메시지를 전송한다. 이에 대응되는 스럽이 서버에도 존재하여 서버 측 스럽이 메시지를 수신한 후 적절한 서버의 프로시저를 호출한다. 필요한 경우 반환 값들도 동일한 방식으로 다시 되돌려 준다. Windows 시스템에서는 스럽 코드 Microsoft Interface Definition Language (MIDL)로 작성된 명세로부터 컴파일된다. 이 언어는 클라이언트와 서버 프로그램 사이의 인터페이스를 정의하는 데 사용된다.

매개변수 정돈(parameter marshalling)은 클라이언트와 서버 기기의 데이터 표현방식의 차이 문제를 해결한다. 32비트 정수를 예로 들어보자. 어떤 기계는 최상위 바이트(most-significant byte)를 먼저 저장하고(big-endian), 어떤 기계(little-endian)는 최하위 바이트(least-significant byte)를 먼저 저장한다. 둘 중 어느 쪽이 “낫다"고 말할 수 없으며 컴퓨터 구조의 선택이다. 이와 같은 차이를 해결하기 위해 대부분의RPC 시스템은 기종 중립적인 데이터 표현 방식을 정의한다. 이러한 표현 방식 중 하나가 XDR (external data representation)이다. 클라이언트 측에서는 서버에게 데이터를 보내기 전 매개변수 정돈 작업의 일환으로 전송할 데이터를 기종 중립적인 XDR 형태로 바꾸어서 보낸다. 수신측 기계에서는 XDR 데이터를 받으면 매개변수를 풀어내면서 자기 기종의 형태로 데이터를 바꾼 후 서버에게로 넘겨준다.

또 다른 중요한 문제는 호출의 의미에(semantic) 관한 것이다. 지역 프로시저 호출의 경우 극단적인 경우에만 실패하지만 RPC 경우는 네트워크 오류 때문에 실패할 수도있고, 메시지가 중복되어 호출이 여러 번 실행될 수도 있다. 이 문제를 해결하는 방법은운영체제가 메시지가 최대 한 번 실행되는 것이 아니라 정확히 한번 처리되도록 보장하게 하는 것이다. 대부분의 지역 프로시저 호출은 “정확히 한 번”의 기능성을 가지고 있으나 이를 구현하는 것은 더 어렵다.

우선 "최대 한 번"을 고려하자. 이 의미는 각 메시지에 타임스템프를 매기는 것으로보장할 수 있다. 서버는 이미 처리한 모든 메시지의 타임스템프 기록을 가지거나 중복된메시지를 검사해 낼 수 있을 만큼의 기록을 가져야 한다. 기록에 보관된 타임스탬프를 가진 메시지가 도착하면 그 메시지는 무시된다. 이렇게 하면 클라이언트는 한 번 이상 메시지를 보낼 수 있고 메시지에 대한 실행이 단 한 번 실행된다는 것을 보장받을 수 있다.

정확히 한 번"의 의미를 가지려면 서버가 요청을 받지 못하는 위험을 제거할 필요가 있다. 이를 완수하려면 서버는 위에서 설명한 "최대 한 번" 프로토콜을 구현하고 추가로 RPC 요청이 수신되었고 실행됐다는 응낙(acknowledgement) 메시지를 보내야만 한다. 이 ACK 메시지는

네트워킹에서 일반적이다. 클라이언트는 해당 호출에 대한ACK를 받을 때까지 주기적으로 각 RPC 호출을 재전송해야 한다.

또 하나 다루어야 할 중요한 문제는 클라이언트와 서버 간의 통신 문제이다. 일반적인 프로시저 호출의 경우, 바인딩(binding)이라는 작업이 링킹/적재/실행 시점에 행해진다(9장). 이때 프로시저의 이름이 프로시저의 메모리 주소로 변환된다. 이와 마찬가지로 RPC도 클라이언트와 서버의 포트를 바인딩 해야 하는데, 클라이언트는 서버의 포트번호를 어떻게 알 수 있는가? 두 시스템에는 모두 상대방에 대한 완전한 정보가 없다(공유 메모리가 없기 때문에).

이를 위해 보통 두 가지 방법이 사용된다. 한 가지 방법은 고정된 포트 주소 형태로 미리 정해 놓는 방법이다. 컴파일 할 때 RPC에는 이 고정된 포트 번호를 준다. 컴파일 되고 나면 그 후에는 서버가 그 포트 번호를 임의로 바꿀 수 없다. 두 번째는 랑데부방식에 의해 동적으로 바인딩 하는 방법이다. 보통 운영체제는 미리 정해져 있는 고정RPC 포트를 통해 랑데부용 디먼(matchmaker라고 불림)을 제공한다. 그러면 클라이언트가 자신이 실행하기를 원하는 RPC 이름을 담고 있는 메시지를 랑데부 디먼에게 보내서, RPC 이름에 대응하는 포트 번호가 무엇인지 알려달라고 요청한다. 그러면 포트번호가 클라이언트에게 반환되고, 클라이언트는 그 포트 번호로 RPC 요청을 계속 보낸다(시스템이 crash 되거나 그 프로세스가 종료되지 않는 한). 이 방식은 통신 초기에 오버헤드가 좀 들기는 하지만 첫 번째 방식보다 더 유연하다. 그림 3.29가 본보기 상호 작용을 보인다.

RPC는 분산 파일 시스템(distributed file system, DFS)을 구현하는 데 유용하다(19장). DFS는 몇 개의 RPC daemon과 클라이언트로 구현할 수 있다. 메시지가 원격지 DFS 서버 포트로 보내지고 이 서버는 file operation을 실행해 준다. 메시지는 실행할 디스크 연산을 포함한다. 이 디스크 연산은 아마도 보통의 파일 관련 연산에 해당하는read (), write (), rename (), delete (), status () 등일 것이다. DFS는 이 연산 결과를 클라이언트에게 반환 메시지로 보낸다. 예를 들면, 어떤 메시지는 파일 전체를 보내라는 것일 수 있고, 어떤 메시지는 몇 개의 블록만 보내라는 것이 될 수 있다. 후자의 경우는 그것을 몇 차례 반복하면 파일 전체를 보내게 될 수도 있게 된다.

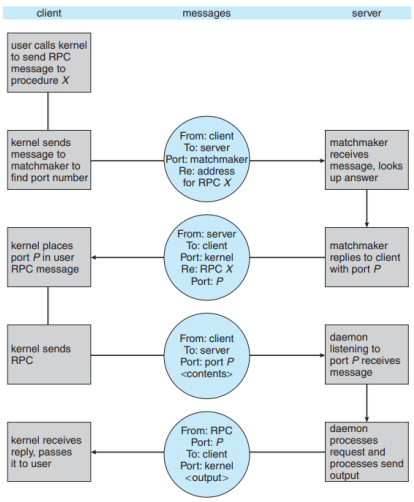


Figure 3.23 Execution of a remote procedure call (RPC).

[그림 3.23]

3.6.2.1. Android RPC

RPC는 일반적으로 분산 시스템에서 클라이언트-서버 컴퓨팅과 관련되어 있지만 동일한 시스템에서 실행되는 프로세스 간 IPC의 형태로 사용될 수도 있다. Android 운영체제는 바인더 프레임워크에 포함된 풍부한 IPC 기법의 집합을 가지고 있는데, 이 중RPC는 프로세스가 다른 프로세스의 서비스를 요청할 수 있게 한다.

Android는 응용 프로그램 구성요소를 Android 응용 프로그램에 유용성을 제공하는 기본 빌딩 블록으로 정의하며, 앱은 여러 응용 프로그램 구성요소를 결합하여 필요한기능을 구현할 수 있다. 이러한 응용 프로그램 구성요소 중 하나는 사용자 인터페이스가없지만 백그라운드로 실행되며 장기 실행 연산을 실행하거나 원격 프로세스에 대한 작업을 수행하는 서비스이다. 서비스의 예로는 백그라운드에서 음악을 재생하고 다른 프로세스 대신 네트워크 연결을 통해 데이터를 검색하여 데이터를 다운로드 할 때 다른 프로세스가 실행 중단되는 것을 방지할 수 있다. 클라이언트 앱이 bindService () 서비스메소드를 호출하면 해당 서비스가 "바운드" 되어 메시지 전달 또는 RPC를 사용하여 클라이언트-서버 통신을 제공할 수 있다.

바운드 서비스는 Android 클래스 Service를 상속해야 하며 클라이언트가 bindService ()를 호출할 때 호출되는 onBind () 메소드를 구현해야 한다. 메시지 전달의경우 onBind () 메소드는 Messenger 서비스를 반환하며, 이 서비스는 클라이언트에서서비스로 메시지를 보내는 데 사용된다. Messenger 서비스는 단방향 통신만 가능하다.서비스가 클라이언트에게 응답을 보내야 하는 경우, 클라이언트도 Messenger 서비스를 제공해야 하며 서비스로 보내진 Message 객체의 replyTo 필드에 포함되어 제공된다. 그런 다음 서비스는 클라이언트에게 메시지를 보낼 수 있다.

RPC를 제공하기 위해 onBind () 메소드는 클라이언트가 서비스와 상호 작용하기위해 사용하는 원격 오브젝트 안에 메소드를 정의하는 인터페이스를 반환해야 한다. 이인터페이스는 일반 Java 구문으로 작성되며 AIDL (Android Interface Definition Language)을 사용하여 스텝 파일을 생성한다. 이 스럽 파일이 원격 서비스에 대한 인터페이스 역할을 수행한다.

여기에서는 AIDL과 바인더 서비스를 사용하여 remoteMethod () 라는 일반 원격 서비스를 제공하는 과정을 간략하게 설명한다. 원격 서비스의 인터페이스는 다음과같다.

```
/* RemoteService.aidl */
interface RemoteService
{
boolean remoteMethod (int x, double y);
}
```

이 파일은 RemoteService.aidl로 저장된다. Android 개발 키트는 이를 사용하여aidl 파일에서 .java 인터페이스와 이 서비스의 RPC 인터페이스 역할을 하는 스럽을생성한다. 서버는 .aidl 파일에 의해 생성된 인터페이스를 구현해야 하며 클라이언트가remoteMethod ()를 호출할 때 이 인터페이스의 구현이 호출된다.

클라이언트가 bindService ()를 호출하면 서버에서 onBind() 메소드가 호출되고 RemoteService 객체의 스럽이 클라이언트에 반환된다. 그런 다음 클라이언트는 다음과 같이 원격 메소드를 호출할 수 있다.

```
RemoteService service;
        ...
service.remoteMethod (3, 0.14);
```

내부적으로 Android 바인더 프레임 워크는 매개변수 정돈을 처리하고, 프로세스사이에 정돈된 매개변수를 전송하고, 필요한 서비스 구현을 호출하고 반환 값을 클라이언트 프로세스에 보낸다.

3.6.3. Pipes

파이프는 두 프로세스가 통신할 수 있게 하는 전달자로서 동작한다. 파이프는 초기UNIX 시스템에서 제공하는 IPC 기법의 하나였다. 파이프는 통상 프로세스 간에 통신하는 더 간단한 방법의 하나이지만 통신할 때 여러 제약을 한다. 파이프를 구현하기 위해서는 다음 4가지 문제를 고려해야 한다.

1. 파이프가 단방향 통신 또는 양방향 통신을 허용하는가?
2. 양방향 통신이 허용된다면 반이중(half duplex) 방식인가, 전이중(full duplex)방식인가? 반이중 방식은 한순간에 한 방향 전송만 가능하고 전이중 방식은 동시에 양방향 데이터 전송이 가능하다.
3. 통신하는 두 프로세스 간에 부모-자식과 같은 특정 관계가 존재해야만 하는가?
4. 파이프는 네트워크를 통하여 통신이 가능한가, 아니면 동일한 기계 안에 존재하는두 프로세스끼리만 통신할 수 있는가?

4. Threads

3장에서 소개된 프로세스 모델은 프로세스가 단일 제어 스레드를 가진 실행 프로그램이라고 가정했다. 그러나 모든 최신 운영 체제는 프로세스가 여러 개의 제어 스레드를 포함하여 작동한다. 이 장에서는 Pthreads, Windows 및 Java 스레드 라이브러리를 위한 API에 대한 설명과 멀티 스레드 컴퓨터 시스템과 관련된 많은 개념을 소개한다. 또한 멀티 스레드 프로그래밍과 관련된 여러 가지 문제와 운영 체제 설계에 미치는 영향을 소개한다. 마지막으로 윈도우즈 및 리눅스 운영 체제가 커널 수준에서 스레드를 지원하는 방법을 소개한다.

4.1. Overview

스레드는 프로그램 내에서 CPU를 사용하는 기본 단위이다. 스레드 ID, 프로그램 카운터, 레지스터 세트, 스택으로 구성된다. 코드, 데이터 및 열려 있는 파일, 프로세스 신호와 같은 다른 운영 체제 리소스들은 동일한 프로세스에 속하는 다른 스레드들과 공유된다. 전통적인 프로세스 또는 중요한 기능을 수행하는 일부 프로세스들은 단일 스레드만 제어한다. 프로세스가 여러 개의 스레드들을 제어하고 있는 경우 한 번에 둘 이상의 태스크를 수행할 수 있다.

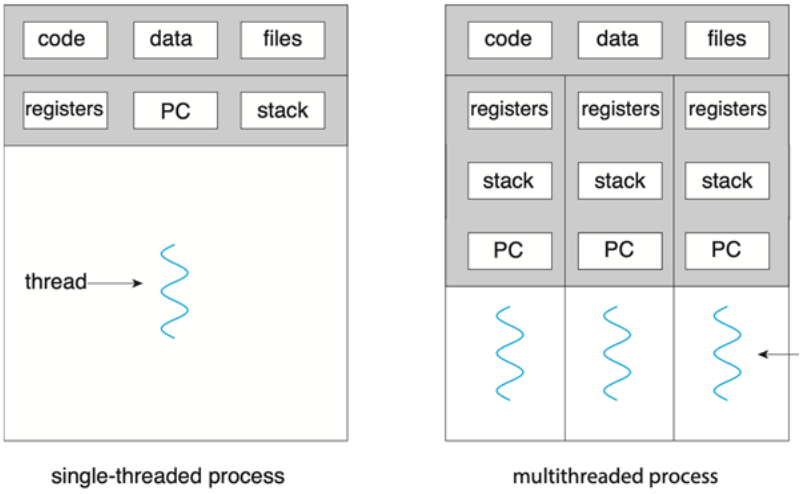
4.1.1. Motivation

최신 컴퓨터에서 실행되는 대부분의 소프트웨어 응용 프로그램은 멀티스레드로 동작한다. 응용 프로그램은 일반적으로 여러 개의 스레드를 가진 별도의 프로세스로 구현된다. 예를 들어 웹 브라우저는 동시에 이미지 또는 텍스트를 표시하는 스레드, 네트워크에서 데이터를 검색하는 스레드를 실행할 수 있다. 워드 프로세서는 그래픽을 표시하는 스레드, 사용자의 키 입력에 응답하는 스레드, 백그라운드에서 철자 및 문법 검사를 수행하는 스레드를 실행할 수 있다. 멀티코어 시스템에서 다수의 코어의 처리 기능을 활용하도록 애플리케이션을 설계할 수도 있는데, 이러한 애플리케이션은 여러 컴퓨팅 코어에 걸쳐 여러 CPU 집약적인 작업을 병렬로 수행할 수 있다.

특정 상황에서는 여러 가지 유사한 작업을 수행하기 위해 단일 애플리케이션이 필요할 수 있습니다. 예를 들어, 웹 서버는 웹 페이지, 이미지, 음성 파일 등에 대한 클라이언트 요청을 처리한다. 트래픽이 높은 웹 서버는 동시에 최대 수천 개의 클라이언트가 액세스할 수 있다. 웹 서버가 기존의 단일 스레드 프로세스로 실행되면 한 번에 하나의 요청만 처리할 수 있으며 해당 요청을 전송한 클라이언트 외의 클라이언트들은 요청이 처리될 때까지 매우 오랜 시간을 기다려야 할 것이다.

한 가지 해결책은 서버가 요청을 수락하는 단일 프로세스로 실행되도록 하는 것이다. 서버는 요청을 수신하면 해당 요청을 처리하는 별도의 프로세스를 생성한다. 스레드 방식이 대중화되기 전에는 일반적으로 이러한 방식이 사용되었다. 그러나 프로세스 생성은 시간이 많이 걸리고 리소스가 많이 소모되는 문제가 있었다. 새로운 프로세스가 기존 프로세스와 동일한 작업을 수행할 경우, 오버헤드로 인해 낮은 효율을 보였다. 일반적으로 여러 스레드가 포함된 하나의 프로세스를 사용하는 것이 더 효율적이다. 웹 서버 프로세스가 멀티스레드인 경우, 서버는 클라이언트 요청을 수신하는 별도의 스레드를 생성한다. 요청을 받으면 서버는 다른 프로세스를 만드는 대신 새로운 스레드를 만들어 요청을 처리하고 추가 요청 수신을 기다린다.

아래 그림은 싱글 스레드와 멀티 스레드의 작동 방식을 나타낸다. 프로세스 내의 스레드들은 코드, 데이터, 열린 파일 등의 영역을 공유한다.

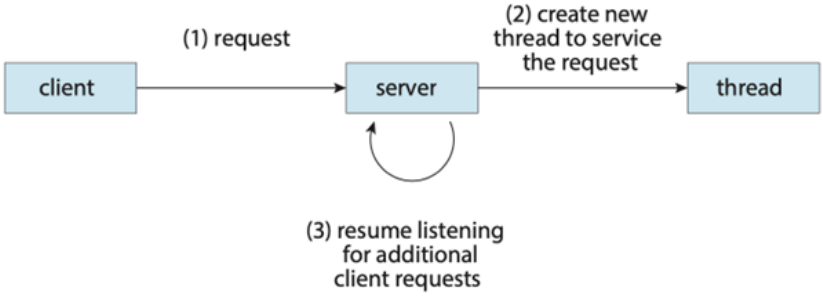


[싱글 스레드와 멀티 스레드의 작동 방식]

일반적인 함수, 프로시저 호출과 유사한 통신 메커니즘을 제공함으로써 프로세스 간 통신을 중재한다. 일반적으로 RPC 서버는 멀티스레드로 동작한다. 서버는 메시지를 수신할 때 별도의 스레드를 사용하여 메시지를 처리한다. 이렇게 하면 서버는 여러 개의 동시 요청을 처리할 수 있다.

마지막으로, 대부분의 운영 체제 커널은 이제 멀티 스레드로 동작한다. 여러 스레드가 커널에서 작동하며, 각 스레드는 장치 관리, 메모리 관리, 인터럽트 처리와 같은 특정 작업을 수행한다. 예를 들어, 솔라리스의 커널에는 인터럽트 처리를 위한 스레드 집합이 있으며, 리눅스는 시스템의 빈 메모리 영역을 관리하기 위해 커널 스레드를 사용한다.

아래 그림은 멀티스레드 방식을 사용한 서버의 구조를 나타낸다. 스레드를 생성해 요청 핸들링을 맡긴다.



[멀티스레드 방식을 사용한 서버의 구조]

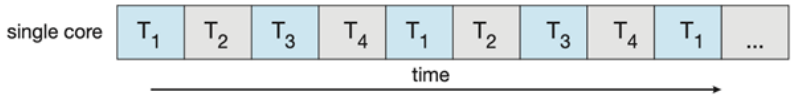
4.1.2. Benefits

멀티스레딩 방식의 장점은 크게 아래의 네 가지로 분류할 수 있다.

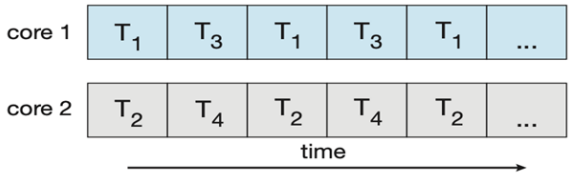
1. 반응성 : 대화형 프로그램을 멀티스레딩 방식으로 구현하면 프로그램의 일부가 차단되거나 긴 작업을 수행하는 경우에도 프로그램이 다른 작업들을 계속 실행되도록 함으로써 사용자에 대한 응답성을 높일 수 있다. 이 특징은 사용자 인터페이스를 설계할 때 매우 유용하다. 예를 들어 사용자가 버튼을 클릭하면 시간이 많이 걸리는 작업을 수행할 때 어떤 일이 발생하는지 생각해 볼 수 있다. 단일 스레드 응용 프로그램은 작업이 완료될 때까지 사용자에게 응답하지 않는다. 반면에, 시간이 많이 걸리는 작업을 별도의 스레드에서 수행하는 경우, 응용 프로그램은 다른 스레드들을 통해 사용자에게 반응할 수 있다.
2. 리소스 공유: 프로세스는 공유 메모리 및 메시지 전달과 같은 방식을 통해서만 리소스를 공유할 수 있다. 이러한 방식은 프로그래머가 명시적으로 구현해야 한다. 그러나 스레드는 기본적으로 자신이 속한 프로세스의 메모리 및 리소스를 공유한다. 코드 및 데이터 공유의 이점은 응용 프로그램이 동일한 주소 공간 내에서 여러 개의 서로 다른 스레드를 실행시킬 수 있다는 것이다.
3. 경제성 : 프로세스 생성을 위해 메모리 및 리소스를 할당하는 데는 비용이 많이 든다. 스레드는 자신이 속한 프로세스의 리소스를 공유하기 때문에 스레드를 생성하고 컨텍스트 전환을 하는 것이 프로세스 간 컨텍스트 전환보다 더 경제적이다. 오버헤드의 차이를 경험적으로 알아낼 수는 없지만, 일반적으로 스레드보다 프로세스를 만들고 관리하는 데 훨씬 더 많은 시간이 소요된다. 예를 들어 Solaris에서 프로세스를 만드는 것은 스레드를 만드는 것보다 약 30배 느리고 컨텍스트 전환은 약 5배 느리다.
4. 확장성 : 멀티스레드의 이점은 스레드가 서로 다른 프로세서 코어에서 병렬로 실행될 수 있는 멀티프로세서 아키텍처에서 훨씬 더 크게 작용한다. 단일 스레드 프로세스는 사용 가능한 프로세서 수에 관계없이 하나의 프로세서에서만 실행할 수 있다.

4.2. Multicore Programming

컴퓨터 디자인 역사 초기에 더 많은 컴퓨터 성능을 이끌어내는 방법으로 단일 CPU 시스템은 다중 CPU 시스템으로 발전했다. 최근 CPU 설계의 방향은 하나의 칩에 여러 개의 컴퓨팅 코어를 배치하는 것이다. 각 코어는 운영 체제에 별도의 프로세서로 표시된다. 코어가 여러 개의 CPU 칩에 걸쳐 나타나든, 단일 CPU 칩 내에 나타나든, 우리는 이러한 시스템을 멀티 코어 또는 멀티 프로세서 시스템이라고 부른다. 멀티 스레드 프로그래밍은 이러한 다중 컴퓨팅 코어의 보다 효율적인 사용과 향상된 동시적 실행을 위한 메커니즘을 제공한다. 예를 들어 스레드가 4개인 애플리케이션을 생각해 볼 수 있다. 단일 컴퓨팅 코어가 있는 시스템에서 동시성은 처리 코어가 한 번에 하나의 스레드만 실행할 수 있기 때문에 시간이 지남에 따라 스레드의 실행이 지연된다는 것을 의미한다. 그러나 여러 개의 코어가 있는 시스템에서 동시성은 각 코어에 별도의 스레드를 할당할 수 있기 때문에 스레드가 병렬로 실행될 수 있다.



주목해야 할 것은 병렬성과 동시성의 차이이다. 시스템은 물리적으로 둘 이상의 작업을 동시에 수행할 수 있는 경우 병렬적이다. 이와는 대조적으로, 동시 시스템은 모든 작업이 진행되도록 함으로써 둘 이상의 작업을 진행한다. 따라서 병렬화 없이 동시성을 가질 수 있다. SMP와 멀티코어 아키텍처가 등장하기 전에 대부분의 컴퓨터 시스템은 단일 프로세서만 가지고 있었다. CPU 스케줄러는 시스템에서 프로세스 간에 빠르게 전환함으로써 병렬적으로 실행된다고 착각하도록 설계되었으며, 따라서 모든 프로세스가 실행될 수 있도록 한다. 이러한 프로세스는 동시에 실행되었지만 병렬로 실행되는 것은 아니다. 시스템이 수십 개의 스레드에서 수천 개의 스레드로 성장하면서 CPU 설계자들은 스레드 성능을 개선하기 위해 하드웨어를 추가함으로써 시스템 성능을 향상시켰다. 현대의 인텔 CPU는 코어당 2개의 스레드를 지원하는 반면 오라클 T4 CPU는 코어당 8개의 스레드를 지원한다. 이 지원은 고속 전환을 위해 여러 스레드를 코어에 로드할 수 있음을 의미한다. 멀티 코어 컴퓨터는 코어 수와 하드웨어 스레드 지원이 계속 증가할 것이다.



위 그림은 싱글 코어에서 동시성을 구현한 것과 멀티코어에서 병렬적으로 실행되는 과정을 시간에 따라 나타낸 것이다. 동시적으로 실행되는 것은 병렬적으로 실행되고 있다는 착각을 일으킬 수도 있다.

- Amdahl's Law

Amdahl의 법칙은 비병렬 컴포넌트와 병렬 컴포넌트를 모두 가진 응용 프로그램에 컴퓨팅 코어를 추가함으로써 얻을 수 있는 잠재적인 성능 이득을 확인하는 공식이다. S가 N개의 처리 코어가 있는 시스템에서 연속적으로 수행되어야 하는 응용 프로그램의 일부분일 경우 공식은 다음과 같다.

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

위 공식에서 주목할 점은 N이 무한대에 가까워질수록 속도가 1/S로 수렴한다는 것이다. 예를 들어 애플리케이션의 40%가 직렬로 수행되는 경우, 우리가 추가하는 프로세싱 코어 수에 관계없이 최대 속도 향상은 2.5배이다. 이것은 Amdahl의 법칙 뒤에 있는 기본 원칙이다. 애플리케이션의 직렬 부분이 컴퓨팅 코어를 추가하여 얻는 성능에 불균형적인 영향을 미칠 수 있다.
어떤 사람들은 Amdahl의 법칙이 현대 멀티코어 시스템의 설계에 사용되는 하드웨어 성능 향상을 고려하지 않는다고 주장한다. 이러한 주장은 현대 컴퓨터 시스템에서 처리 코어의 수가 계속 증가함에 따라 Amdahl의 법칙이 더 이상 적용되지 않을 수 있음을 의미한다.

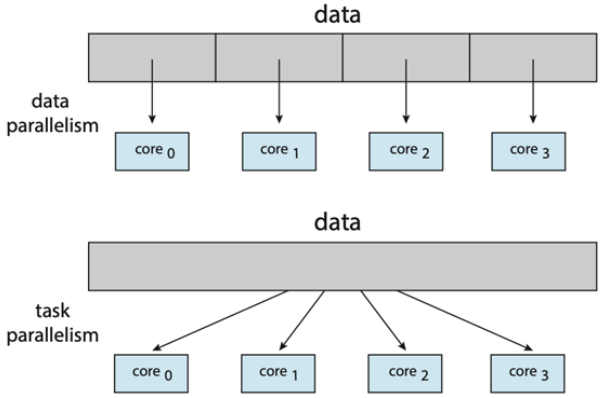
4.2.1. Programming Challenges

다중 코어 시스템으로의 추세는 시스템 설계자와 애플리케이션 프로그래머에게 다중 코어 설계를 더 잘 활용하라는 압력을 계속 가하고 있다. 운영 체제 설계자는 병렬 실행을 허용하기 위해 다중 처리 코어를 사용하는 스케줄링 알고리즘을 작성해야 한다. 응용 프로그램 프로그래머의 경우 멀티 스레드화된 새 프로그램을 설계해야 할 뿐만 아니라 기존 프로그램을 수정하고 보완하는 것이 과제를 안게 되었다. 일반적으로 멀티코어 시스템의 프로그래밍의 과제에는 다음과 같은 다섯 가지 영역이 있다.

1. 태스크 파악. 여기에는 프로세스들을 검토하여 별도의 동시 작업으로 나눌 수 있는 영역을 찾는 작업이 포함된다. 작업이 서로 독립적이므로 개별 코어에서 병렬로 실행되는 것이 가장 이상적이다.
2. 균형. 프로그래머는 병렬로 실행할 수 있는 태스크를 식별하면서 작업이 동일한 가치의 동일한 작업을 수행하도록 보장해야 한다. 경우에 따라 특정 작업을 실행하는 것이 다른 작업만큼 전체 프로세스에 많은 가치를 제공하지 못할 수도 있다. 별도의 코어를 사용하여 해당 작업을 실행하는 것은 비용 측면에서 가치가 없거나 손해일 수 있다.
3. 데이터 분할. 애플리케이션을 별도의 작업으로 나누듯 해당 작업에서 액세스하고 조작하는 데이터를 별도의 코어로 나눠야 한다.
4. 데이터 종속성. 작업에서 액세스하는 데이터는 두 개 이상의 작업 간의 종속성을 검사해야 합니다. 한 작업이 다른 작업의 데이터에 의존할 때, 프로그래머는 작업 실행이 데이터 종속성을 수용하도록 동기화되어야 한다.
5. 테스트 및 디버깅. 프로그램이 여러 코어에서 병렬로 실행될 때, 많은 다른 실행 경로가 가능하다. 이러한 동시적으로 실행되는 프로그램을 테스트하고 디버깅하는 것은 본질적으로 싱글 스레드 응용 프로그램을 테스트하고 디버깅하는 것보다 더 어렵다.

이러한 과제들 때문에, 많은 소프트웨어 개발자들은 멀티코어 시스템의 출현은 미래에 소프트웨어 시스템을 설계하기 위한 완전히 새로운 접근방식을 필요로 할 것이라고 주장한다.

아래 그림은 데이터 또는 태스크(프로세스 또는 스레드)를 병렬화 처리하는 방식을 나타낸다. 데이터 병렬화는 각 코어가 다수의 자료들을 처리하는 과정이다



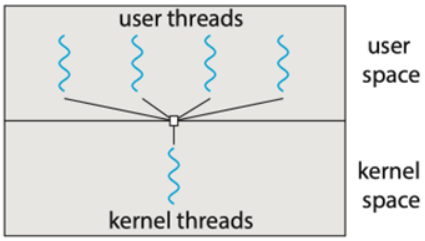
[데이터 병렬화 처리 방식 시각화]

4.3. Multithreading Models

지금까지는 일반적인 의미의 스레드를 다루었다. 그러나 스레드에 대한 지원은 사용자 레벨, 사용자 스레드에 대한 지원 또는 커널 스레드에 대한 커널에 의해 제공될 수 있다. 사용자 스레드는 커널 위에서 지원되며 커널 지원 없이 관리되지만 커널 스레드는 운영 체제에서 직접 지원 및 관리된다. 윈도우, 리눅스, 맥 OS X, 솔라리스 등 거의 모든 현대의 운영 체제는 커널 스레드를 지원한다. 따라서 사용자 스레드와 커널 스레드 사이에 관계가 존재해야 한다. 이 장에서는 이러한 관계를 설정하는 세 가지 일반적인 방법, 즉 다대일 모델, 일대일 모델 및 다대다 모델을 살펴본다.

4.3.1. Many-to-One Model

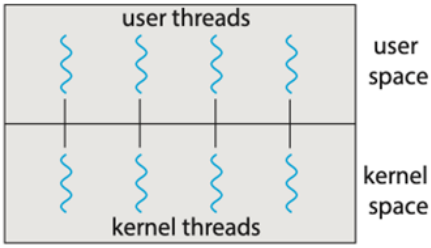
다대일 모델은 많은 사용자 레벨 스레드를 하나의 커널 스레드에 매핑한다. 스레드 관리는 사용자 공간에서 스레드 라이브러리에 의해 수행되므로 효율적이다. 그러나 스레드가 차단 시스템 호출을 하면 전체 프로세스가 실행이 정지된다. 또한 한 번에 하나의 스레드만 커널에 액세스할 수 있기 때문에 다중 스레드는 다중 코어 시스템에서 병렬로 실행될 수 없다. 여러 개의 프로세싱 코어를 활용할 수 없기 때문에 이 모델을 계속 사용하는 시스템은 거의 없다.



[Many-to-One Model]

4.3.2. One-to-One Model

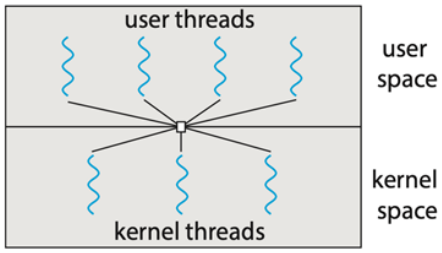
일대일 모델은 각 사용자 스레드를 각각의 커널 스레드에 매핑한다. 스레드가 차단 시스템 호출을 할 때 다른 스레드가 실행되도록 함으로써 다대일 모델보다 더 많은 동시성을 제공한다. 또한 다중 프로세서에서 여러 스레드를 병렬로 실행할 수 있다. 이 모델의 유일한 단점은 사용자 스레드를 만들려면 해당 커널 스레드를 만들어야 한다는 것이다. 커널 스레드를 만드는 오버헤드는 응용 프로그램의 성능에 부담을 줄 수 있기 때문에 이 모델의 대부분의 구현은 시스템에서 지원하는 스레드 수를 제한한다. 리눅스는 윈도우 운영 체제 계열과 함께 일대일 모델을 구현한다.



[One-to-One Model]

4.3.3. Many-to-Many Model

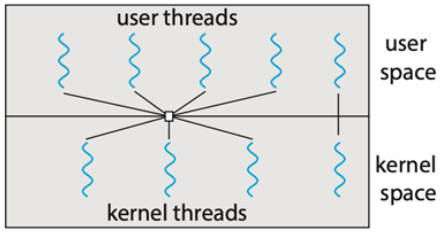
다대 다 모델은 많은 사용자 레벨 스레드를 더 적거나 같은 수의 커널 스레드로 다중화한다. 커널 스레드의 수는 특정 응용 프로그램이나 특정 머신에 따라 정해져 있을 수 있다. 멀티프로세서 위에서는 프로그램은 더 많은 커널 스레드를 할당받을 수 있다.



[Many-to-Many Model]

이 설계는 동시성에 영향을 미친다. 다대일 모델은 개발자가 원하는 만큼 많은 사용자 스레드를 만들 수 있게 하는 반면, 커널은 한 번에 하나의 스레드만 스케줄 할 수 있기 때문에 실질적인 동시적 실행을 하지 않는다. 일대일 모델은 더 큰 동시성을 허용하지만, 개발자는 응용프로그램 내에 너무 많은 스레드를 만들지 않도록 주의해야 한다. 개발자는 필요한 만큼 사용자 스레드를 생성할 수 있으며, 해당 커널 스레드는 멀티프로세서에서 병렬로 실행될 수 있다. 또한 스레드가 차단 시스템 호출을 수행할 때 커널은 실행을 위해 다른 스레드에 스케줄링 될 수 있다.

2-수준 모델로 명명된 다대 다 모델의 한 변형은 여전히 많은 사용자 레벨 스레드를 더 적거나 같은 수의 커널 스레드에 다중화하지만 어떤 사용자 레벨 스레드를 커널 스레드에 일대일로 바인딩할 수 있게 한다.



[Another version of Many-to-Many Model]

4.4. Thread Libraries

스레드 라이브러리는 프로그래머에게 스레드를 만들고 관리하기 위해 탄생했다. 스레드 라이브러리를 구현하는 방법은 크게 두 가지이다. 첫 번째 접근법은 커널 지원 없이 사용자 공간에서 전적으로 라이브러리를 제공하는 것이다. 라이브러리에서 함수를 호출하면 시스템

호출이 아닌 사용자 공간에서 로컬 함수 호출이 발생한다. 두 번째 접근법은 운영 체제에서 직접 지원하는 커널 수준의 라이브러리를 구현하는 것이다. 이 경우, 라이브러리의 코드와 데이터 구조는 커널 공간에 존재한다. 라이브러리를 위한 API의 함수를 호출하면 일반적으로 커널에 대한 시스템 호출이 발생한다.

현재 POSIX Pthreads, Windows, Java의 세 가지 주요 스레드 라이브러리가 사용되고 있다. POSIX 표준의 스레드 확장인 Pthreads는 사용자 수준 또는 커널 수준 라이브러리로 제공된다. 윈도우 스레드 라이브러리는 윈도우 시스템에서 사용할 수 있는 커널 레벨 라이브러리이다. 자바 스레드 API는 자바 프로그램에서 스레드를 직접 만들고 관리할 수 있게 해준다. 그러나 대부분의 경우 JVM이 호스트 운영 체제 위에서 실행되기 때문에 자바 스레드 API는 일반적으로 호스트 시스템에서 사용할 수 있는 스레드 라이브러리를 사용하여 구현된다. 이는 윈도우 시스템에서 자바 스레드는 일반적으로 윈도우 API를 사용하여 구현되며, 유닉스, 리눅스, 맥OS 시스템은 일반적으로 Pthread를 사용한다.

POSIX와 윈도우 스레딩의 경우, 전역적으로 선언된 모든 데이터, 즉 함수 외부에 선언된 데이터는 동일한 프로세스에 속하는 모든 스레드 간에 공유된다. 자바에는 글로벌 데이터에 대한 동등한 개념이 없기 때문에 공유 데이터에 대한 접근은 스레드 간에 명시적으로 배치되어야 한다.

비동기 스레딩과 동기 스레딩은 멀티스레딩을 구현하는 전략이다. 비동기 스레딩의 경우 부모가 자식 스레드를 생성하면 부모 스레드가 실행을 재개하여 부모 스레드와 자식 스레드가 동시에 독립적으로 실행된다. 스레드는 독립적이기 때문에 일반적으로 스레드 간에 데이터 공유가 거의 없다. 비동기 스레딩은 멀티 스레드 서버에서 사용되는 전략이며, 응답성이 높은 사용자 인터페이스를 설계하기 위해 일반적으로 사용된다.

동기 스레딩은 상위 스레드가 하나 이상의 하위 스레드를 만든 다음 모든 하위 스레드가 종료될 때까지 기다려야 다시 시작할 수 있다. 여기서 부모에 의해 작성된 스레드는 동시에 작업을 수행하지만, 부모에 의해 이 작업이 완료될 때까지 계속할 수 없습니다. 각 스레드가 작업을 마치면 종료되고 부모 스레드와 결합된다. 모든 하위 항목이 종료한 후에만 상위 항목이 실행을 재개할 수 있습니다. 일반적으로 동기 스레딩은 스레드 간에 상당한 데이터 공유를 포함합니다. 예를 들어, 상위 스레드는 다양한 하위 스레드에 의해 계산된 결과를 결합할 수 있다.

4.4.1. Pthreads

Pthreads는 스레드 생성 및 동기화를 위한 API를 정의하는 POSIX 표준을 의미한다. 이것은 스레드 동작에 대한 규격이지 구현한 것이 아니다. 운영 체제 설계자는 원하는 방식으로 사양을 구현할 수 있다. 수많은 시스템들이 Pthreads 규격을 구현하고 있으며, 대부분은 리눅스와 mac OS를 포함한 유닉스 계열 시스템이다. Windows에서는 기본적으로 Pthreads를 지원하지 않지만 Windows용 일부 타사 구현을 사용할 수 있다.

4.4.2. Windows Threads

Windows 스레드 라이브러리를 사용하여 스레드를 만드는 기술은 여러 가지 면에서 Pthreads 기술과 유사하다.

4.4.3. Java Threads

스레드는 자바 프로그램에서 프로그램 실행의 기본 모델이며, 자바 언어의 API는 스레드의 생성과 관리를 위한 풍부한 기능 세트를 제공한다. 모든 자바 프로그램은 적어도 하나의 제어 스레드로 구성되며, 심지어 main() 메서드로 구성된 단순한 자바 프로그램도 JVM에서 하나의 스레드로 실행된다. 자바 스레드는 윈도우, 리눅스, mac OS를 포함한 JVM을 제공하는 모든 시스템에서 사용할 수 있다. 자바 스레드 API는 안드로이드 애플리케이션에서도 사용할 수 있다.

4.5. Implicit Threading

멀티코어 프로세싱의 지속적인 성장과 함께 수백 또는 수천 개의 스레드를 포함하는 애플리케이션이 대중적으로 개발되고 있다. 그러나 이러한 응용 프로그램을 설계하는 것은 쉬운 일이 아니다. 프로그래머는 이전에 설명된 과제뿐만 아니라 추가적인 어려움도 해결해야 한다.

이러한 어려움을 해결하고 멀티 스레드 응용 프로그램의 설계를 더 잘 지원하는 한 가지 방법은 응용 프로그램 개발자에서 컴파일러 및 런타임 라이브러리로 스레딩의 생성과 관리를 이전하는 것이다. 암묵적 스레딩이라고 불리는 이 전략은 오늘날 인기 있는 트렌드다. 여기서부터는 암묵적 스레딩을 통해 멀티 코어 프로세서를 활용할 수 있는 멀티 스레드 프로그램을 설계하기 위한 세 가지 대안적 접근 방식을 서술한다.

4.5.1. Thread Pools

다중 스레드 웹 서버 설계에서는 서버는 요청을 수신할 때마다 요청을 처리할 별도의 스레드를 생성한다. 별도의 스레드를 만드는 것이 별도의 프로세스를 만드는 것보다 확실히 우수하지만, 그럼에도 불구하고 멀티 스레드 서버는 잠재적인 문제를 가지고 있다. 첫 번째 문제는 스레드를 만드는 데 필요한 시간과 스레드가 작업을 완료하면 폐기된다는 것이다. 두 번째 문제는 더 골치아프다. 모든 동시 요청이 새 스레드에서 서비스되도록 허용하는 경우 시스템에서 동시에 활성화된 스레드 수에 제한을 두지 않는다. 무제한 스레드는 CPU 시간이나 메모리와 같은 시스템 리소스를 소모하게 된다. 이 문제에 대한 한 가지 해결책은 스레드 풀을 사용하는 것이다.

스레드 풀의 기본 아이디어는 프로세스 시작 시 여러 스레드를 생성하고 풀 안에 배치하는 것이다. 풀에서 스레드는 작업을 할당받기를 기다린다. 서버는 요청을 수신하면 이 풀에서 스레드가 사용 가능한 경우 스레드를 활성화하고 서비스 요청을 전달한다. 스레드가 서비스를 완료하면 풀로 돌아가 다음 작업을 기다립니다. 풀에 사용 가능한 스레드가 없는 경우, 서버는 스레드가 사용 가능해질 때까지 기다린다.

스레드 풀은 다음과 같은 장점들을 가진다.

- 기존 스레드로 요청을 처리하는 것이 스레드를 생성하기 위해 기다리는 것보다 빠르다
- 스레드 풀은 동시에 존재할 수 있는 스레드 수를 제한한다. 이는 많은 수의 동시 스레드를 지원할 수 없는 시스템에서 특히 중요하다.
- 수행할 작업을 작업 생성 메커니즘과 분리하면 작업을 실행하기 위해 서로 다른 전략을 사용할 수 있다. 예를 들어 작업이 시간 지연 후 실행되거나 주기적으로 실행되도록 예약할 수 있다.

풀의 스레드 수는 시스템의 CPU 수, 물리적 메모리 양 및 예상되는 동시 클라이언트 요청 수와 같은 요소에 따라 휴리스틱하게 설정할 수 있다. 보다 정교한 스레드 풀 아키텍처는 사용 패턴에 따라 풀의 스레드 수를 동적으로 조정할 수 있다. 이러한 아키텍처는 시스템의 부하가 낮을 때 더 작은 풀, 더 적은 메모리 소비량을 가지는 추가적인 이점이 있다.

4.5.2. OpenMP

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

int main(int argc, char *argv[])
{
   /* sequential code */

   #pragma omp parallel
   {
      printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}
```

OpenMP는 컴파일러 명령어 집합이며 공유 메모리 환경에서 병렬 프로그래밍을 지원하는 C, C++ 또는 FORTRAN으로 작성된 프로그램을 위한 API이다. OpenMP는 병렬로 실행할 수 있는 코드 블록을 식별한다. 응용 프로그램 개발자들은 컴파일러 명령어를 그들의 코드에 병렬로 삽입하고, 이 명령어들은 OpenMP 런타임 라이브러리가 영역을 병렬로 실행하도록 지시한다. 아래 C 프로그램은 printf() 문을 포함하는 병렬 영역 위의 컴파일러 명령어를 보여준다:

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

OpenMP가 #pragma omp parallel 지시어를 만나면, 시스템에 프로세싱 코어가 있는 만큼 많은 스레드를 생성한다. 따라서 듀얼 코어 시스템의 경우 2개의 스레드가 생성되고, 쿼드 코어 시스템의 경우 4개의 스레드가 생성된다. 그런 다음 모든 스레드가 동시에 병렬 영역을 실행한다. 각 스레드가 병렬 영역을 벗어나면 종료된다.

OpenMP는 병렬화 루프를 포함하여 코드 영역을 병렬로 실행하기 위한 몇 가지 추가 명령어를 제공한다. 예를 들어, 크기가 N인 배열 a와 b가 두 개 있다고 가정하겠다. 우리는 그들의 내용을

종합해서 결과를 배열 c에 넣기를 원한다. 다음 코드 세그먼트를 사용하여 이 작업을 병렬로 실행할 수 있으며, 여기에는 루프를 병렬화하기 위한 컴파일러 명령이 포함되어 있다.

OpenMP는 #pragma omp parallel for 지시어를 만나면 for 루프 안의 작업들을 스레드별로 나누어 실행한다.

OpenMP는 병렬화를 위한 지시문을 제공할 뿐만 아니라 개발자가 여러 병렬화 수준 중에서 선택할 수 있도록 한다. 예를 들어, 수동으로 스레드 수를 설정할 수 있다. 또한 개발자는 데이터가 스레드 간에 공유되는지 또는 스레드에 대해 비공개인지 식별할 수 있다. OpenMP는 리눅스, 윈도우, 맥 OS X 시스템용 여러 오픈 소스 및 상용 컴파일러에서 사용할 수 있다.

4.5.3. Grand Central Dispatch

그랜드 센트럴 디스패치(GCD)는 애플의 맥 OS X와 iOS 운영 체제를 위한 기술로서, 응용 프로그램 개발자들이 병렬로 실행할 코드의 부분을 식별할 수 있도록 하는 C 언어, API, 런타임 라이브러리의 확장 기능을 결합한 것이다. OpenMP와 마찬가지로 GCD는 스레딩의 대부분의 세부 사항을 관리한다.

GCD는 블록으로 알려진 C와 C++ 언어의 확장 기능을 식별한다. 블록은 단순히 일의 자체적인 단위이다. 이 값은 한 쌍의 중괄호 앞에 삽입된 캐럿 문자로 지정된다. 다음은 블록의 간단한 예이다.

```
^{ printf("I am a block"); }
```

GCD는 런타임 실행을 위한 블록을 디스패치 큐에 배치하여 예약한다. 큐에서 블록을 제거하면 관리하는 스레드 풀에서 사용 가능한 스레드에 블록을 할당한다. GCD는 연속 및 동시 디스패치 큐의 두 가지 유형을 식별한다.

연속 대기열에 배치된 블록은 FIFO 순서대로 제거된다. 큐에서 블록을 제거한 후에는 다른 블록을 제거하기 전에 실행을 완료해야 한다. 각 프로세스에는 자체 연속 대기열(또는 대기열이라고 함)이 있다. 개발자는 특정 프로세스에 로컬인 추가 연속 대기열을 만들 수 있다. 연속 대기열은 여러 작업의 순차적 실행을 보장하는 데 유용하다.

동시 큐에 배치된 블록도 FIFO 순서로 제거되지만, 한 번에 여러 블록을 제거할 수 있으므로 여러 블록을 병렬로 실행할 수 있다. 시스템 전체의 동시 디스패치 대기열에는 낮음, 기본값 및 높음이라는 세 가지 우선 순위에 따라 구분된다. 우선 순위는 블록의 상대적 중요도에 대한 근사치를 나타낸다. 간단히 말해서 우선 순위가 높은 블록은 우선 순위가 높은 디스패치 큐에 배치되어야 한다.

Swift 프로그래밍 언어의 경우, 작업은 클로저를 사용하여 정의되며, 이는기능 단위를 표현한다는 점에서 블록과 유사하다. 구문적으로 Swift 클로저는 선행 캐럿을 뺀 블록과 동일한 방식으로 작성된다.

다음의 Swift 코드는 기본 우선순위의 동시 큐와 dispatch_async()함수를 사용하여 코드 블록을 큐에 추가하는 동작을 나타낸 것이다.

```
let queue = dispatch_get_global_queue
   (QOS_CLASS_USER_INITIATED, 0)

dispatch_async(queue,{ print("I am a closure.") })
```

내부적으로 GCD의 스레드 풀은 POSIX 스레드로 구성되어 있다. GCD는 적극적으로 풀을 관리하여 애플리케이션 수요와 시스템 용량에 따라 스레드 수를 늘리고 축소할 수 있다.

4.5.4. Other Approaches

스레드 풀, OpenMP 및 Grand Central Dispatch는 멀티 스레드 애플리케이션을 관리하기 위한 수많은 새로운 기술 중 일부이다. 다른 상업적 접근 방식에는 인텔의 TBB(Threading Building Block)와 마이크로소프트의 여러 제품과 같은 병렬 및 동시 라이브러리가 포함된다. 자바 언어와 API도 동시 프로그래밍을 지원하는 방향으로 다양한 움직임을 보이고 있다. 주목할 만한 예로는 암시적 스레드 생성 및 관리를 지원하는 java.util.concurrent 패키지가 있다.

4.6. Threading Issues

이 장에서는 멀티스레드 프로그램을 디자인할 때 고려해야 할 사항들에 대해 기술한다.

4.6.1. The fork() and exec() System Calls

3장에서는 포크() 시스템 호출을 사용하여 별도의 중복 프로세스를 생성하는 방법을 설명했다. 다중 스레드 프로그램에서 포크() 및 exec() 시스템 호출의 의미는 달라진다

프로그램의 한 스레드가 fork()를 호출하는 경우, 새 프로세스는 모든 스레드를 복제할까, 아니면 새 프로세스는 단일 스레드로서 복제될지에 혼란이 생길 수 있다. 일부 유닉스 시스템은 모든 스레드를 복제하는 버전과 포크() 시스템 호출을 호출하는 스레드만 복제하는 두 가지 버전의 포크() 방식을 채택했다.

exec() 시스템 호출은 일반적으로 3장에서 설명한 것과 동일한 방식으로 작동한다. 즉, 스레드가 exec() 시스템 호출을 호출하는 경우 exec()에 대한 매개 변수에 지정된 프로그램이 모든 스레드를 포함한 전체 프로세스를 대체한다.

두 가지 버전의 포크() 중 어떤 것을 사용할지는 응용 프로그램에 따라 다르다. exec()이 포크 직후에 호출되면 exec()에 지정된 프로그램이 프로세스를 대체하므로 모든 스레드를 복제할 필요가 없다. 이 경우 호출 스레드만 복제하는 것이 적절하다. 그러나 포크 후 별도의 프로세스가 exec()을 호출하지 않는 경우, 별도의 프로세스는 모든 스레드를 복제해야 한다.

4.6.2. Signal Handling

UNIX 시스템에서는 특정 이벤트가 발생했음을 프로세스에 알리기 위해 신호를 사용한다. 신호는 이벤트의 근원과 이벤트가 신호되는 이유에 따라 동기 또는 비동기적으로 수신될 수 있다. 동기 신호든 비동기 신호든 모든 신호는 동일한 패턴을 따른다.

1. 신호는 특정 이벤트가 발생할 때 생성된다.
2. 신호는 프로세스로 전달된다.
3. 신호가 전달되면 신호는 반드시 처리되어야 한다.

동기 신호의 예로는 잘못된 메모리 액세스와 0으로 나누기 오류 등이 있다. 실행 중인 프로그램이 이러한 작업 중 하나를 수행하면 신호가 생성된다. 동기 신호는 신호를 발생시킨 연산을 수행한 동일한 프로세스로 전달되므로, 동기적으로 처리된다.

실행 중인 프로세스 외부의 이벤트에 의해 신호가 생성되면 해당 프로세스는 신호를 비동기적으로 수신한다. 이러한 신호의 예로는 특정 키 입력(ctrl + c)으로 프로세스를 종료하는 것 또는 타이머가 만료되는 것이 있습니다. 일반적으로 비동기 신호는 다른 프로세스로 전송된다.

신호는 다음 두 가지 중 하나의 처리기에 의해 처리될 수 있다.

1. 기본 신호 처리기
2. 사용자 정의 신호 처리기

모든 신호에는 해당 신호를 처리할 때 커널이 실행하는 기본 신호 처리기가 있다. 기본 신호 처리기의 동작은 신호를 처리하기 위해 호출되는 사용자 정의 신호 처리기에 의해 재정의될 수 있다. 신호는 다양한 방식으로 처리된다. 일부 신호(예: 창 크기 변경)는 간단히 무시되고, 다른 신호(예: 잘못된 메모리 액세스)는 프로그램을 종료함으로써 처리된다.

단일 스레드 프로그램에서 신호를 처리하는 것은 간단하다. 신호는 항상 프로세스로 전달된다. 그러나 다중 스레드 프로그램에서 신호 전달은 더 복잡하며, 프로세스에는 여러 스레드가 있을 수 있다. 신호 전달 체계가 필요해졌다. 일반적으로 다음과 같은 옵션이 있다.

1. 신호가 적용되는 스레드에 신호를 전달한다.
2. 프로세스의 모든 스레드에 신호를 전달한다.
3. 프로세스의 특정 스레드에 신호를 전달한다.
4. 프로세스에 대한 모든 신호를 수신할 특정 스레드를 할당한다.

신호를 전달하는 방법은 생성되는 신호의 유형에 따라 달라진다. 예를 들어 동기 신호는 신호를 발생시키는 스레드에 전달되어야 하며 프로세스의 다른 스레드에 전달되지 않아야 한다. 그러나

비동기 신호의 상황은 항상 명확하지는 않다. 프로세스를 종료하는 신호(예: control + c)와 같은 일부 비동기 신호는 모든 스레드로 전송되어야 한다.

UNIX에서 신호를 전달하는 기본 함수는 다음과 같다.

kill(pid_t pid, int signal)

이 함수는 특정 신호가 전달될 프로세스(pid)를 지정한다. 대부분의 멀티스레드 버전의 UNIX에서는 스레드가 받아들일 신호와 차단할 신호를 지정할 수 있다. 따라서 어떤 경우에는 비동기 신호가 차단되지 않는 스레드에만 전달될 수 있다. 그러나 신호는 한 번만 처리하면 되기 때문에 일반적으로 신호를 차단하지 않는 첫 번째 스레드로만 전달된다. POSIX pthreads는 지정된 스레드(tid)로 신호를 전달할 수 있는 다음과 같은 기능을 제공한다.

pthread_kill(pthread_t tid, int signal)

Windows는 신호에 대한 지원을 명시적으로 제공하지 않지만, 비동기 프로시저 호출(APC)을 사용하여 신호를 모방할 수 있다. APC 기능을 사용하면 사용자 스레드가 특정 이벤트에 대한 알림을 수신할 때 호출될 함수를 지정할 수 있다. APC는 유닉스의 비동기 신호와 거의 같다. 그러나 유닉스는 멀티 스레드 환경에서 신호를 처리하는 방법을 논의해야 하는 반면, APC 기능은 프로세스보다는 특정 스레드로 전달되기 때문에 APC 기능이 더 간단하다고 할 수 있다.

4.6.3. Thread Cancellation

스레드 취소에는 스레드가 완료되기 전에 스레드를 종료하는 작업이 포함된다. 예를 들어, 여러 스레드가 동시에 데이터베이스를 검색 중이고 한 스레드가 결과를 반환하는 경우 나머지 스레드가 취소될 수 있다. 사용자가 웹 브라우저에서 단추를 누르면 웹 페이지가 더 이상 로드되지 않는다. 웹 페이지는 여러 스레드를 사용하여 로드되는 경우가 많다. 각 이미지는 별도의 스레드에 로드된다. 사용자가 브라우저에서 중지 버튼을 누르면 페이지를 로드하는 모든 스레드가 취소된다.

취소되는 스레드를 대상 스레드라고 명명한다. 대상 스레드의 취소는 두 가지 다른 시나리오에서 발생할 수 있다.

1. 비동기 취소. 하나의 스레드가 대상 스레드를 즉시 종료한다.
2. 지연 취소. 대상 스레드는 정기적으로 종료 여부를 확인하여 질서 있는 방식으로 자신을 종료할 수 있는 기회를 제공한다.

취소는 취소된 스레드에 리소스가 할당되었거나 스레드가 다른 스레드와 공유하는 동안 스레드가 취소된 상황에서 발생한다. 문제는 비동기 취소 시 발생하게 된다. 운영 체제는 취소된 스레드에서 시스템 리소스를 회수하지만 리소스를 전부 회수하지는 않는다. 따라서 비동기적으로 스레드를 취소해도 필요한 시스템 전체 리소스를 확보하지 못할 수 있다.

반면 지연 취소의 경우 한 스레드는 대상 스레드를 취소해야 함을 나타내지만 취소는 대상 스레드가 플래그를 확인하여 취소 여부를 결정한 후에만 취소된다. 스레드는 안전하게 취소할 수 있는 지점 및 시점에서 이 검사를 수행할 수 있다.

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

 . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Pthreads에서 스레드 취소는 pthread cancel() 함수를 사용하는 것으로 시작된다. 대상 스레드의 식별자가 함수에 대한 매개 변수로 전달된다. 다음 코드는 스레드를 만든 다음 취소하는 동작을 수행한다.

pthread cancel()을 호출하면 대상 스레드를 취소하라는 요청만 표시되며 실제 취소는 대상 스레드가 요청을 처리하도록 설정된 방식에 따라 달라진다. Pthread는 세 가지 취소 모드를 지원한다. 각 모드는 표에 표시된 바와 같이 상태 및 유형으로 정의된다. 스레드는 API를 사용하여 취소 상태를 설정하고 유형을 설정할 수 있다.

표에서 알 수 있듯이, Pthreads는 스레드가 취소를 비활성화하거나 활성화하도록 상태를 조정한다. 취소가 비활성화되어 있으면 스레드를 취소할 수 없는 상태를 나타낸다. 그러나 취소 요청은 보류 상태로 남아 있으므로 스레드는 나중에 취소를 활성화하고 요청에 응답할 수 있다.

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

위 표는 pthread에서 스레드의 취소/종료 상태를 나타낸다.

기본 취소 유형은 지연 취소이다. 여기서 취소는 스레드가 취소 지점에 도달한 경우에만 발생합니다. 취소 지점을 설정하는 방법 중 하나는 pthread test cancel() 함수를 호출하는 것이다. 취소 요청이 보류 중인 것으로 확인되면 청소 처리기로 알려진 기능이 호출된다. 이 기능을 사용하면 스레드가 종료되기 전에 스레드가 획득한 모든 리소스를 해제할 수 있다.

다음 코드는 스레드가 지연 취소를 사용하여 취소 요청에 응답한다.

```
while (1) {
   /* do some work for awhile */

    . . .

   /* check if there is a cancellation request */
   pthread_testcancel();
}
```

앞에서 설명한 문제 때문에 Pthreads documentation에서는 비동기 취소를 권장하지 않는다. 리눅스 시스템에서는 스레드 취소가 Pthreads API를 사용하는 신호를 통해 처리된다.

4.6.4. Thread-Local Storage

어떤 프로세스에 속하는 스레드들은 프로세스의 데이터를 공유한다. 이러한 데이터 공유는 멀티스레드 프로그래밍의 이점 중 하나이다. 그러나 상황에 따라 각 스레드에 특정 데이터의 복사본이 필요할 수 있다. 이러한 데이터를 스레드 로컬 스토리지(TLS)라고 부른다. 예를 들어, 트랜잭션 처리 시스템에서 각 트랜잭션을 별도의 스레드로 처리할 수 있다. 또한 각 트랜잭션에는 고유한 식별자가 할당될 수 있다. 각 스레드를 고유한 식별자와 연결하기 위해 스레드 로컬 스토리지를 사용할 수 있다.

TLS와 로컬 변수는 유사하나 별개의 개념이다. 로컬 변수는 단일 함수 호출 중에만 볼 수 있는 반면, TLS 데이터는 스레드 내에서 호출되는 모든 함수에서 볼 수 있다. 어떤 면에서 TLS는 정적 데이터와 유사하다. 차이점은 TLS 데이터는 각 스레드에 고유하다는 것이다. 윈도우와 Pthreads를 포함한 대부분의 스레드 라이브러리는 스레드 로컬 스토리지에 접근할 수 있도록 하는 기능을 지원한다.

4.6.5. Scheduler Activations

다중 스레드 프로그램에서 고려해야 할 마지막 문제는 커널과 스레드 라이브러리 간의 통신에 관한 것으로, 4.3.3에서 논의된 다대 다 및 2단계 모델을 사용함으로써 필요할 수 있다. 여기에서 다룰 이 조정을 통해 커널 스레드의 수를 동적으로 조정하여 최상의 성능을 보장할 수 있다.

다대다 또는 2단계 모델을 구현하는 많은 시스템들은 사용자와 커널 스레드 사이를 매개하는 중간 자료구조를 배치한다. 일반적으로 경량 프로세스 또는 LWP는 응용 프로그램이 실행할 사용자 스레드를 예약할 수 있는 가상 프로세서처럼 동작한다. 각 LWP는 커널 스레드에 연결되며, 운영 체제가 물리적 프로세서에서 실행되도록 스케줄링하는 커널 스레드이다. 커널 스레드가 차단되면(예: I/O 작업이 완료될 때까지 대기), LWP도 차단된다. LWP의 상단 레벨에 부착된 사용자 레벨 스레드도 차단된다.

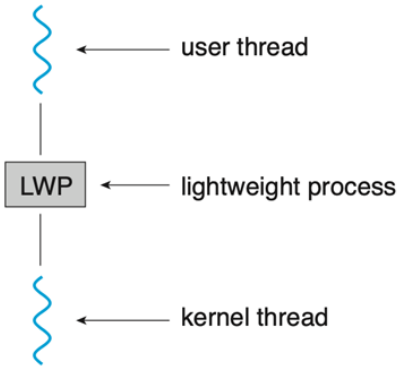
응용 프로그램은 효율적으로 실행되기 위해 몇 가지 LWP를 요구할 수 있다. 단일 프로세서에서 CPU 바인딩된 애플리케이션을 실행한다고 가정해 보겠다. 이 시나리오에서는 한 번에 하나의 스레드만 실행할 수 있으므로 하나의 LWP로 충분하다. 그러나 I/O 집약적인 애플리케이션은 여러 LWP를 실행해야 할 수 있다. 일반적으로 각 동시 차단 시스템 호출에 대해 LWP가 필요하다. 예를 들어, 다섯 개의 서로 다른 파일 읽기 요청이 동시에 발생한다고 가정해 보겠다.

모두 커널에서 입출력 완료를 기다리고 있을 수 있기 때문에 5개의 LWP가 필요하다. 프로세스가 4개의 LWP만 가지고 있다면, 다섯 번째 요청은 커널에서 LWP 중 하나가 반환될 때까지 기다려야 한다.

사용자 스레드 라이브러리와 커널 간의 통신을 위한 한 가지 방식은 스케줄러 활성화라고 알려져 있다. 커널은 응용 프로그램에 일련의 가상 프로세서(LWP)를 제공하며 응용 프로그램은 사용 가능한 가상 프로세서에 사용자 스레드를 스케줄링할 수 있다. 또한 커널은 특정 이벤트를 응용 프로그램에 알려야 한다. 이 절차를 업콜이라고 한다. 업콜은 스레드 라이브러리에서 업콜 처리기를 사용하여 처리되며 업콜 처리기는 가상 프로세서에서 실행되어야 한다. 업콜을 트리거하는 이벤트 중 하나는 응용 프로그램 스레드가 자신의 실행을 차단하려고 할 때 발생한다. 이 시나리오에서 커널은 스레드가 막으려는 것을 알리고 특정 스레드를 식별하기 위해 응용 프로그램에 업콜을 한다. 그런 다음 커널은 새로운 가상 프로세서를 애플리케이션에 할당한다. 응용 프로그램은 이 새 가상 프로세서에서 업콜 처리기를 실행하여 차단 스레드의 상태를 저장하고 차단 스레드가 실행 중인 가상 프로세서를 포기한다. 그런 다음 업콜 처리기는 새 가상 프로세서에서 실행할 수 있는 다른 스레드를 예약한다. 차단 스레드가 대기하고 있던 이벤트가 발생하면 커널은 스레드 라이브러리에 다른 업콜을 하여 이전에 차단된 스레드가 이제 실행 가능한 상태임을 알려준다.

이 이벤트에 대한 업콜 처리기도 가상 프로세서가 필요하며 커널은 새로운 가상 프로세서를 할당하거나 사용자 스레드 중 하나를 비워 놓고 가상 프로세서에서 업콜 처리기를 실행할 수 있다. 차단되지 않은 스레드를 실행 가능으로 표시한 후 응용 프로그램은 사용 가능한 가상 프로세서에서 실행되도록 스레드를 예약한다.

아래 사진은 유저 스레드와 커널 스레드를 중개하는 LWP의 연결 구조를 나타낸다.



[LWP 연결구조]

4.7. Operating-System Examples

이 장에서는 윈도우즈 및 리눅스 시스템에서 스레드가 구현되는 방법을 예시를 통해 설명하였다.

4.7.1. Windows Threads

윈도우즈 응용 프로그램은 별도의 프로세스로 실행되며 각 프로세스에는 하나 이상의 스레드가 포함된다. 또한 Windows는 4.3.2에 설명된 일대일 매핑을 사용한다. 여기서 각 사용자 레벨 스레드는 연결된 커널 스레드에 매핑된다.

스레드의 일반적인 구성 요소는 다음과 같다.

- 스레드를 고유하게 식별하는 ID
- 프로세서의 상태를 나타내는 레지스터 세트
- 프로그램 카운터
- 스레드가 사용자 모드에서 실행 중일 때 사용되는 사용자 스택과 스레드가 커널 모드에서 실행 중일 때 사용되는 커널 스택
- 다양한 런타임 라이브러리 및 DLL(Dynamic Link Library)에서 사용하는 개인 저장소 영역

레지스터 세트, 스택 및 개인 저장소 영역을 스레드의 컨텍스트(문맥) 라고 한다.

스레드의 주요 데이터 구조는 다음과 같다

- ETHREAD — 실행 스레드 블록
- KTHREAD — 커널 스레드 블록
- TEB — 스레드 환경 블록

ETHREAD의 주요 구성 요소에는 스레드가 속한 프로세스에 대한 포인터와 스레드가 제어를 시작하는 루틴의 주소가 포함된다. ETHREAD에는 해당 KTHREAD에 대한 포인터가 포함되어 있다.

KTREAD에는 스레드에 대한 스케줄링 및 동기화 정보가 포함되어 있다. 또한 KTHREAD는 커널 스택(스레드가 커널 모드에서 실행 중일 때 사용됨)과 TEB에 대한 포인터를 포함한다.

ETHREAD와 KTHREAD는 전적으로 커널 공간에 존재하며, 이는 커널만이 접근할 수 있다는 것을 의미한다. TEB는 스레드가 사용자 모드에서 실행될 때 액세스하는 사용자 공간 자료구조입니다. TEB에는 스레드 식별자, 사용자 모드 스택 및 스레드 로컬 스토리지를 위한 배열이 포함되어 있다.

4.7.2. Linux Threads

리눅스는 3장에서 설명한 것처럼 프로세스를 복제하는 전통적인 기능을 갖춘 fork() 시스템 호출을 제공한다. 리눅스는 clone() 시스템 호출을 사용하여 스레드를 생성하는 기능도

제공한다. 그러나 리눅스는 프로세스와 스레드를 구분하지 않는다. 실제로 리눅스는 프로그램 내의 제어 흐름을 언급할 때 프로세스나 스레드 대신 태스크라는 용어를 사용한다.

clone()이 호출되면 상위 태스크와 하위 태스크 간에 얼마나 많은 공유를 수행할지 결정하는 일련의 플래그가 전달된다. 예를 들어 clone()이 [CLONE FS, CLONE VM, CLONE SIGHAND, CLONE FILES] 를 전달했다고 가정하자. 그러면 상위 및 하위 태스크는 동일한 파일 시스템 정보, 동일한 메모리 공간, 동일한 신호 처리기 및 동일한 열린 파일 집합을 공유한다. 이 방식으로 clone()을 사용하는 것은 부모 태스크가 하위 태스크와 대부분의 리소스를 공유하기 때문에 이 장에서 설명하는 스레드를 생성하는 것과 같다. 그러나 clone()이 호출될 때 이러한 플래그가 설정되지 않으면 공유가 이루어지지 않으므로 fork() 시스템 호출에서 제공하는 기능과 유사한 기능이 발생한다.

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

[clone()함수에 전달 가능한 플래그들의 예시]

위 표는 clone()함수에 전달될 수 있는 플래그들의 예시 및 그 작동 방식을 나타낸다.

리눅스 커널에서 작업이 표현되는 방식 덕분에 다양한 수준의 공유가 가능하다. 시스템의 각 작업에 대해 고유한 커널 데이터 구조(특히 struct task_struct)가 존재한다. 이 자료 구조는 작업을 위한 데이터를 저장하는 대신 이러한 데이터가 저장되는 다른 자료 구조(예: 열린 파일, 신호 처리 정보 및 가상 메모리 목록을 나타내는 데이터)에 대한 포인터를 포함합니다. fork()가 호출되면 부모 프로세스의 모든 관련 자료 구조의 복사본과 함께 새 작업이 생성된다. clone() 시스템 호출을 수행할 때도 새 태스크가 생성되지만 새 작업은 모든 데이터 구조를 복사하는 대신 clone()으로 전달된 플래그 집합에 따라 상위 작업의 데이터 구조를 가리킨다.

clone() 시스템 호출의 유연성은 컨테이너의 개념으로 확장될 수 있다. 컨테이너는 운영 체제에서 제공하는 가상화 기술로서 서로 격리되어 실행되는 단일 리눅스 커널에서 여러 개의 리눅스 시스템(컨테이너)을 생성할 수 있다. clone()에 전달된 특정 플래그가 부모 태스크와 자식 태스크 간의 데이터 공유량에 따라 태스크가 프로세스 또는 스레드처럼 동작하는지 구별할 수 있는 것처럼, clone() 함수에 전달할 수 있는 다른 플래그를 사용해 리눅스 컨테이너를 만들 수 있다.

5. CPU Scheduling

CPU 스케줄링은 멀티프로그래밍 운영 체제의 기본이다. 프로세스 간에 CPU를 전환함으로써 OS는 컴퓨터를 더 생산적으로 만들 수 있다. 기본적인 CPU 스케줄링 개념 및 몇 가지 CPU 스케줄링 알고리즘을 알아보며 특정 시스템에 대해 적절한 알고리즘을 선택하는 방법을 알아보자.

5.1. Basic Concepts

멀티프로그래밍의 목적은 CPU 활용도utilization를 극대화하기 위해 항상 프로세스가 실행되게 하는것이다. 보통의 프로세스는 일반적으로 I/O 요청이 완료될 때까지 대기하였다가 그가 완료된 후 실행된다. 이 때, CPU는 유휴 상태이므로 많은 낭비를 초래한다. 동시에 여러 프로세스가 메모리에 올라왔을 때, 한 프로세스가 대기하면 OS는 해당 프로세스에서 CPU를 빼앗아 다른 프로세스에 CPU를 제공한다. 그리고 이 패턴이 계속되면 이를 멀티프로그래밍이라고 한다. 컴퓨터 자원의 낭비를 막기 위해서는 멀티프로그래밍을 사용하여 하나의 프로세스가 대기해야 할 때마다 다른 프로세스가 CPU의 사용을 인계할 수 있어야 한다.

5.1.1. CPU –I/O Burst Cycle

프로세스의 실행은 CPU 실행 및 I/O 대기 주기로 구성된다. 프로세스는 CPU실행과 입출력 대기라는 두 상태 사이를 번갈아 가며 실행된다. CPU가 메모리와 데이터를 주고받으면서 일하는것을 CPU 버스트라 하고, 디스크로부터 데이터를 주고받는것을 I/O 버스트라고 한다. 프로세스가 실행될 때는 CPU 버스트로 시작되고 그 다음에는 I/O 버스트, 그 다음에는 또 다른 CPU 버스트, 그리고 또 다른 I/O 버스트 등이 반복해서 뒤따른다. 결국, 최종 CPU 버스트는 시스템의 종료 요청terminate request으로 끝이난다.

종류가 다른 두 프로세스에 동일한 CPU 시간을 주어 동급으로 처리해야 하는가? 아니다. 시간과 자원의 낭비를 막기 위해 같은 시간이 주어지면 안된다. 이들은 구분해서 처리 할 필요가 있는데 그를 위해서는 스케줄링이 필수적이다.

5.1.2. CPU Scheduler

CPU가 유휴 상태가 될 때마다 운영 체제는 실행 대기열ready queue에 있는 프로세스 중 하나를 선택해야만 한다. 선택된 프로세스는 단기 스케줄러 또는 CPU 스케줄러에 의해 수행된다. 스케줄러는 메모리의 프로세스들 중 실행할 준비가 된 프로세스를 선택하고 해당 프로세스에 CPU를 할당한다. Ready queue가 반드시 선입선출 FIFO 되진 않는다. ready queue는 FIFO queue, priority queue, tree 또는 unordered linked list 등으로 구현될 수 있다. 그러나 개념적으로 ready queue에 있는 모든 프로세스는 CPU에서 실행될 기회를 기다리며 줄을 선다. 대기열의 레코드는 일반적으로 프로세스의 프로세스 제어 블록 process control blocks (PCB)이다.

5.1.3. Preemptive Scheduling

CPU 스케줄링 결정은 다음 네 가지 상황에서 이루어질 수 있다.

1. 프로세스 상태가 running -> waiting 전환되는 경우 (예: I/O 요청 또는 자식 프로세스 종료 wait() 호출 결과)
2. 프로세스 상태가 running -> ready 전환되는 경우 (예: 인터럽트 interrupt 발생 시)
3. 프로세스 상태가 waiting -> ready 전환되는 경우 (예: I/O 완료 시)
4. 프로세스가 종료 termination 될 때

1번과 4번의 상황의 경우엔 스케줄링 방법의 선택 옵션이 없고 무조건 대기열에 있는 새로운 프로세스를 선택해야만 하는 상황이다. 그러나 2번과 3번 상황에는 선택할 옵션이 있다. 스케줄링이 상황 1과 4에서만 이루어질 때 스케줄 체계가 비선제적 nonpreemptive 이거나 협조적 cooperative 이라고 말하고 그렇지 않은 경우에는 선제적 preemptive 이라고 말한다.

비선제적 스케줄링에서 CPU가 프로세스에 할당되면 프로세스는 종료 또는 waiting 상태로 전환하여 CPU의 상태가 자유로워질때까지 유지하게 된다. cooperative 스케줄링은 선제적 스케줄링처럼 특별한 하드웨어(예를 들어 타이머)를 필요로 하거나 하지 않기 때문에 특정 하드웨어 플랫폼에서 사용할 수 있는 유일한 방법이다.

여러 프로세스 간에 데이터가 공유될 때 선제적 스케줄링으로 인해 문제가 발생할 수 있다. 데이터의 preemption은 운영 체제 커널의 설계에 영향을 미친다. 시스템 콜을 처리하는 동안 커널은 해당 프로세스를 실행하느라 바쁠 수 있다. 어떠한 활동들은 중요한 커널 데이터를 변경하는 활동일 수 있다. 프로세스가 이런 변경하는 활동의 중간에 CPU를 빼앗고 커널이 동일한 구조를 읽거나 수정해야 한다면 혼돈이 뒤따를 것은 불 보듯 뻔한 일이다. 대부분의 유닉스 버전을 포함한 특정 운영 체제는 컨텍스트 스위칭을 수행하기 전에 시스템 호출이 완료되거나 I/O 블록이 발생할 때까지 기다림으로써 이 문제를 해결한다. 이 체계는 커널 데이터 구조가 일관되지 않은 상태에 있는동안에는 커널이 프로세스를 선점하지 않기 때문에 커널 구조가 단순함을 보장한다. 하지만 이 커널 실행 모델은 주어진 시간 내에 작업을 완료해야 하는 실시간 컴퓨팅 real-time computing을 지원하는 데는 적합하지 않다.

인터럽트interrupt는 언제든 발생할 수 있고 커널에 의해 항상 무시될 수만은 없기 때문에 인터럽트에 의해 영향을 받는 코드의 섹션은 동시에 사용되지 않도록 보호되어야 한다. 운영 체제는 거의 모든 순간 인터럽트를 허용해야 한다. 그렇지 않으면 입력이 손실되거나 출력이 덮어쓰기될 수 있기 때문이다. 이러한 코드 섹션이 여러 프로세스에 의해 동시에 액세스되지 않도록, 입력 시 인터럽트를 비활성화하고 종료 시 인터럽트를 다시 활성화하는 방식으로 진행한다. 인터럽트를 비활성화하는 코드의 섹션은 사실 거의 발생하지 않으며 일반적으로 몇 가지 명령어가 포함되어 있다.

5.1.4. Dispatcher

CPU 스케줄링과 관련된 또 다른 구성 요소는 디스패처dispatcher이다. 디스패처는 단기 스케줄러에 의해 선택된 프로세스에 CPU를 제어하게 해주는 모듈인데 기능은 다음과 같다.

- 컨텍스트 스위칭
- 사용자 모드로 전환
- 프로그램 재시작을 위해 사용자 프로그램의 적절한 위치로 뛰어넘는다.

디스패처는 모든 프로세스 변환 중에 호출되므로 최대한 빠른 스피드를 가져야 한다. 디스패처가 한 프로세스를 중지하고 다른 실행을 시작하는 데 걸리는 시간을 디스패치 지연 시간이라고 한다.

5.2. Scheduling Criteria

특정 상황을 위해 사용할 알고리즘을 선택할 때, 다양한 알고리즘의 속성을 고려해야 한다. 어떤 특성이 비교를 위해 사용되느냐에 따라 어떤 알고리즘이 최선이라고 판단되는지에 상당한 차이가 생길 수 있는데 아래는 그 기준들이다.

- CPU 사용률 Utilization
  높을수록 좋다. 개념적으로 CPU 활용률은 0~100% 범위일 수 있지만, 실제 시스템의 경우 40%에서 90%까지 범위가 지정된다.
- 처리량 Throughput
  정해진 시간동안 얼마나 많은 양의 작업을 수행했는가에 대한 부분이다. 일정 시간 동안 얼마나 많은 프로세스가 처리되었는지에 따라 시간 단위 당 완료되는 프로세스 수를 계산할 수 있다. 긴 프로세스의 경우 이 속도는 한 시간당 하나의 프로세스일 수 있으며, 짧은 프로세스의 경우 일 초당 10개의 프로세스일수도 있다.
- 반환 시간 Turnaround time
  특정 프로세스의 관점에서 중요한 기준은 프로세스를 실행하는 데 걸리는 시간이다. 프로세스 제출 시점부터 완료 시점까지의 간격이 턴어라운드 타임이다. 턴어라운드 타임은 메모리에 들어가기 위해 대기하고, ready queue에서 대기하고, CPU에서 실행하고, I/O를 수행하는 데 소요된 시간의 총 합계이다.
- 대기 시간 Waiting time
  CPU 스케줄링 알고리즘은 프로세스가 실행되거나 I/O를 수행하는 시간에는 영향을 미치지 않는다. 프로세스가 ready queue에 대기하는 시간에만 영향을 미친다. 대기 시간은 ready queue에서 대기한 시간의 합이다.
- 응답 시간 Response time
  요청을 제출한 후 첫 번째 응답 first response 이 생성될 때까지의 시간이다. 응답 시간이라고 하는 이 측정값은 응답을 시작하는 데 걸리는 시간이지 응답을 출력하는 데 걸리는 시간은 아니다. 이 기준에서는 빨리 처리하는것보다도 중요한건 응답이 빠르게 오는것이다.

CPU 활용률 및 처리량을 극대화하고 반환 시간turnaround time, 대기 시간waiting time 및 응답 시간response time을 최소화하는 것이 바람직하다. 많은 경우 평균 측정을 최적화하지만 몇몇 상황에서는 최소값 또는 최대값을 최적화하는 것이 좋을 수도 있다.

예를 들어, 모든 사용자가 좋은 서비스를 받을 수 있도록하려면 최대 응답시간을 최소화해야 할것이다. 데스크탑 시스템과 같이 interactive한 시스템의 경우 평균 속도가 빠르지만 매우 가변적인 시스템보다 응답 시간이 합리적이고 예측 가능한 시스템이 훨씬 바람직하다고 할 수 있다. 하지만 변화폭을 최소화하는 CPU 스케줄링 알고리즘에 대해서는 많은 연구가 진행되지 않았다.

다음 섹션에서는 간단히 설명하기 위해 프로세스당 하나의 CPU 버스트(ms 단위)만 고려하며 비교 척도는 평균 대기 시간을 기준 삼는다.

5.3. Scheduling Algorithms

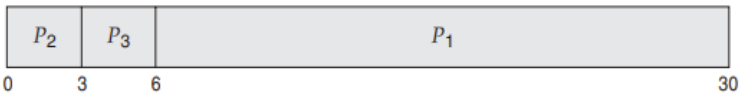
5.3.1. First-Come, First-Served Scheduling

CPU 스케줄링 알고리즘 중 가장 심플한 FCFS(First-Come, First-Served) 스케줄링 알고리즘 체계를 사용하면 CPU를 먼저 요청하는 프로세스가 CPU에 먼저 할당된다. 즉, 선착순 개념이다. 이는 FIFO(First-In-First-Out) 큐를 사용하여 쉽게 관리할 수 있다. 프로세스가 ready queue에 들어가면 PCB는 queue의 마지막 tail 에 연결되는데 CPU가 사용 가능한 경우 대기열 맨 앞에 있는 프로세스에 할당된다. 그리고 실행 중인 프로세스가 queue에서 제거된다. FCFS 스케줄링을 위한 코드는 작성과 이해가 쉽다는 장점이 있지만 FCFS 스케줄링에서의 평균 대기 시간은 종종 꽤 길어진다는 단점도 있다. 아래 예시를 보자.

Process	Burst Time
P_1	24
P_2	3
P_3	3

세 프로세스가 P1, P2, P3 순서로 도착하고 FCFS 스케줄링으로 실행했을 때 평균 대기 시간은 (0 + 24 + 27)/3 = 17ms이며, 아래 간트 Gantt 차트로 각 참여 프로세스의 시작 및 종료 시간을 포함한 특정 일정을 알 수 있다.

[FCFS로 스케줄링 했을때 간트차트]



[버스트시간이 짧은 프로세스부터 실행했을때의 간트차트]

만일 프로세스들이 P2 -> P3 -> P1 순서로 도착하면 결과는 위의 차트와 같다. 이제 평균 대기 시간은 (6 + 0 + 3)/3 = 3ms이다. 이전의 17ms와 비교했을 때 상당한 차이가 있다.

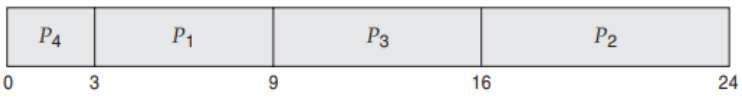
따라서 FCFS 정책의 평균 대기 시간은 일반적으로 최소값이 아니며 프로세스의 CPU 버스트 시간의 차이가 클 경우 평균 대기 시간 또한 상당히 달라질 수 있다. 또한 동적 상황에서의 FCFS 스케줄링의 성능을 고려해본다면 여러 I/O 프로세스가 CPU-bound 프로세스가 완료될 때까지 ready queue에서 대기해야하는 상황이 벌어질 가능성을 생각해 볼 수 있다. 이처럼 다른 모든 프로세스들이 하나의 큰 프로세스가 CPU에서 빠져나오기를 기다리는 것을 호송 효과convoy

effect라 하는데 하나의 느린 프로세스로 인하여 전체 프로세스 세트의 성능이 저하되고 CPU의 낭비로 이어지는 안좋은 상태이다. 이로 인해 CPU 및 디바이스 사용률이 더 짧은 프로세스를 먼저 수행할 수 있는 경우보다 낮아진다. 또한 FCFS 스케줄링 알고리즘은 nonpreemptive하다. CPU가 프로세스에 할당되면 해당 프로세스는 종료 또는 I/O를 요청하여 CPU를 해제할 때까지 CPU를 유지한다. 따라서 FCFS 알고리즘은 각 사용자가 일정한 간격으로 CPU의 공유를 얻는 것이 중요한 시분할 시스템time-sharing systems의 경우 특히 문제가 된다. 한 프로세스가 CPU를 장기간 점유할 수 있도록 허용하는 것은 대단히 효율이 떨어지는 일이다.

5.3.2. Shortest-Job-First Scheduling

SJF 스케줄링 (Shortest-Job-First Scheduling) 알고리즘은 프로세스의 다음 CPU 버스트의 길이를 각 프로세스와 링크시킨다. CPU 사용이 가능해지면 CPU 버스트가 가장 작은 프로세스에 할당된다. 만일 다음으로 들어올 프로세스가 CPU 버스트가 동일한 두 개의 프로세스 일 경우, FCFS 스케줄링을 사용한다. 이 스케줄링은 프로세스의 총 길이보다 다음에 올 CPU 버스트의 길이가 더 중요하기 때문에 이 스케줄링 방법에 더 적합한 용어는 최단 다음 CPU 버스트 알고리즘 shortest-nextCPU-burst algorithm이지만 SJF라는 용어가 훨씬 대중적으로 사용된다. 아래에 있는 SJF 스케줄링의 예시를 살펴보자.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



[SJF스케줄링 간트차트]

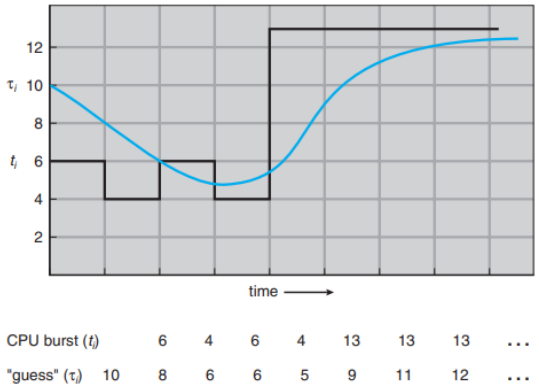
위 차트를 보면, P1의 경우 대기시간이 3ms, P2는 16ms, P3는 9ms, P4는 0ms이며 평균 대기 시간은 (3 + 16 + 9 + 0)/4 = 7ms가 된다. 만약 FCFS 스케줄링을 사용했다면 평균 대기 시간은 10.25ms였을것이다. SJF 스케줄링 알고리즘은 주어진 프로세스 세트에 대한 최소값의 평균 대기 시간을 제공한다는 점에서 최적의 알고리즘으로 입증되었다. 긴 프로세스보다 짧은 프로세스를 먼저 실행하는 것이 효율적이고 결과적으로 평균 대기 시간이 감소하게 된다. 하지만 SJF 알고리즘에서의 문제는 다음 CPU 요청의 길이를 예측해야한다는 것이다. 배치 시스템batch system의 장기(작업) 스케줄링을 위해서 사용자가 작업을 제출할 때 프로세스 시간 제한을 지정하여 사용할 수 있는데 만일 낮은 값을 지정한다면 더 빠른 응답이 가능하긴 하지만 너무 낮은 값은 제한 시간 초과 오류 time-limit-exceeded error를 유발하고 재제출을 요구하기 때문에 사용자는 프로세스 시간 제한을 정확하게 지정하여야 할 것이다.

SJF 스케줄링은 다음에 올 CPU 버스트의 길이를 알 수 없는 단기 스케줄링 수준에서는 구현할 수 없으며 장기 스케줄링에서 자주 사용된다. 하지만 SJF 근사치로 계산한다면 어떨까. 다음 CPU 버스트의 길이는 이전 CPU 버스트와 비슷할 것으로 예측할 수 있기 때문에 다음 CPU 버스트의 길이에 대한 근사치를 계산하면 예측된 CPU 버스트가 가장 짧은 프로세스를 선택할

수 있게된다. 다음 CPU 버스트는 일반적으로 이전 CPU 버스트의 측정된 길이의 지수 평균exponential average으로 예측하는데 다음 공식으로 지수 평균을 정의할 수 있다.

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

*n*번째 CPU 버스트의 길이가 t_n 일때, 예측하고자 하는 것은, τ_{n+1}= 다음 cpu버스트의 예측값이다. 매개 변수는 0보다 크거나 같고 1보다 작거나 같은 수, 즉 0과 1사이에 있는 수 이다. t_n의 값은 가장 최근의 히스토리값이다. 매개 변수는 최근 및 과거 기록의 상대적 가중치를 제어한다. 매개변수 α = 0이면 n+ 1 = n 으로서 최근 히스토리가 영향을 미치지 않는다. α=1이면 n+ 1 = t_n이며, 가장 최근의 CPU 버스트만 문제가 될 것이다. 보통 일반적으로는 α=$\frac{1}{2}$ 이므로 최근 히스토리와 과거 히스토리는 동일하게 가중치를 부여한다. 초기 0은 상수 또는 전체 시스템 평균으로 정의할 수 있다.



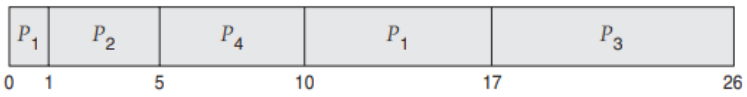
[다음 CPU버스트 길이 예측 그래프]

위의 그래프에서는 매개변수 α = $\frac{1}{2}$ 로 두었고 τ_0 = 10으로 두었다. 위의 그래프를 n+ 1에 대한 공식을 n으로 대체하여 다음과 같이 구할 수 있다.

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n+1} \tau_0.$$

일반적으로 α는 1보다 작기 때문에 (1 - α) 또한 1보다 작으며, 각 연속되는 항은 이전 항보다 가중치가 적다고 볼 수 있다. SJF 알고리즘은 선제적preemptive 알고리즘 일 수도, 비선제적nonpreemptive 알고리즘일 수도 있다. 이전 프로세스가 아직 실행 중일 때 새 프로세스가 ready queue에 도착할 때 두 가지 옵션 중 하나를 선택하게 된다. 새로 도착한 프로세스의 다음 CPU 버스트가 현재 실행 중인 프로세스의 남은 CPU 버스트보다 짧을 때 선제적preemptive SJF 알고리즘은 현재 실행 중인 프로세스를 빼앗아 선점하는 반면, 비선제적 SJF 알고리즘은 현재 실행 중인 프로세스가 CPU 버스트를 끝낼 수 있도록 기다린다. SJF 스케줄링을 이용한 프로세스 실행을 알아보자.

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



[Preemptive SJF스케줄링 간트차트]

P1은 큐에 있는 현재 유일한 프로세스이기 때문에 시간 0에 시작된다. 이후, P2는 시간 1에 도착한다. P1의 남은 시간(7ms)이 P2를 실행하는데 필요한 시간(4ms)보다 길기 때문에 P1이 선점되고 P2가 다음으로 실행된다. 이 예시의 평균 대기 시간은 [(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4 = 26/4 = 6.5ms이다. 비선점적 SJF 스케줄링의 평균 대기 시간을 따진다면 7.75ms가 될 것이다.

5.3.3. Priority Scheduling

SJF 알고리즘은 일반적인 우선순위 스케줄링 알고리즘priority-scheduling algorithm의 특별한 경우이다. 우선 순위는 각 프로세스와 연결되고 CPU는 가장 높은 우선 순위의 프로세스에게 가장 먼저 할당된다. 동일한 우선 순위 프로세스가 있다면 FCFS 순서로 즉, 선착순으로 스케줄링된다. SJF 알고리즘은 우선 순위(p)가 다음 CPU 버스트의 역순인 간단한 우선 순위 알고리즘이었다. CPU 버스트가 클수록 우선 순위가 낮아지고 CPU 버스트가 작을수록 우선 순위가 높아진다. 우선 순위의 높고낮음에 따라 스케줄링의 순서가 결정된다. 우선 순위는 일반적으로 고정된 수의 범위로 표시한다. 0이 가장 높은 우선 순위인지 가장 낮은 우선 순위인지에 대해 정해진 바는 없는데 보통은 0을 가장 높은 우선 순위로 두곤 한다. 아래의 예시에서는 모든 프로세스들이 동시에 도착했다고 가정하고 우선순위 스케줄링을 해본다.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

[Priority 스케줄링 간트차트]

평균 대기 시간은 8.2ms이다. 우선 순위는 내부 또는 외부에서 정의할 수 있는데, 내부적으로 정의한다면 측정 가능한 것 또는 프로세스의 우선순위를 계산할 수 있는 것들을 사용한다. 예를 들어 시간 제한, 메모리 요구 사항, 열려 있는 파일 수, 평균 CPU 버스트 대비 평균 I/O 버스트의 비율 등이 컴퓨팅 우선 순위에 사용된다. 외부 우선 순위는 프로세스의 중요성, 컴퓨터 사용을 위해 지불되는 자금의 종류와 양, 작업을 후원하는 부서, 그리고 종종 정치적 요인들과 같은 운영 체제 외부의 기준에 의해 설정된다. 우선 순위 스케줄링은 SJF 스케줄링과 마찬가지로 선제적일수도, 또는 비선제적일 수도 있다. 새로운 프로세스가 ready queue에 도착하면 현재

실행 중인 프로세스의 우선 순위와 비교한다. 선제적 우선 순위 스케줄링 알고리즘은 새로 도착한 프로세스의 우선 순위가 현재 실행 중인 프로세스의 우선 순위보다 높을 경우 즉시 CPU를 빼앗아온다. 비선점 우선 순위 스케줄링 알고리즘은 새로운 프로세스를 ready queue의 맨 앞에 놓기만 할 것이다. 우선 순위 스케줄링 알고리즘의 주요 문제는 무기한 blocking 또는 starvation이다. 실행할 준비가 되었지만 CPU를 기다리는 프로세스는 차단된 것으로 간주될지도 모른다. 우선 순위 스케줄링 알고리즘은 일부 낮은 우선 순위 프로세스를 무기한 대기 상태로 둘지도 모른다는 큰 문제점이 있다. 프로세스가 많이 대기하고 있는 컴퓨터 시스템에서 고우선순위 프로세스의 지속적인 흐름은 저우선순위 프로세스가 CPU를 얻는 것을 영원히 blocking 할 수 있다. 그에 따라 결국엔 실행되거나, 컴퓨터 시스템이 충돌하여 미완료된 모든 저우선순위 프로세스를 전부 잃게 되는 결말을 맞게 될지도 모르기 때문에 Aging이 필요하다. Aging은 시스템에서 오랫동안 대기하는 프로세스의 우선순위를 점진적으로 증가시키는 것을 의미한다. 이것으로 초기 우선 순위가 최하위인 프로세스도 언젠간 시스템에서 가장 높은 우선 순위를 가지며 실행될 것이다.

5.3.4. Round-Robin Scheduling

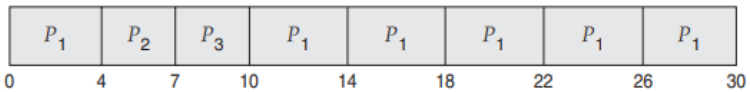
라운드 로빈 Round-Robin Scheduling (RR) 스케줄링 알고리즘은 특히 시간 공유 시스템timesharing systems을 위해 설계되었다. FCFS 스케줄링과 유사하지만 시스템이 프로세스 간에 전환할 수 있도록 선제권preemption이 추가되었다. 타임 퀀텀 또는 타임 슬라이스이라고 불리는 작은 시간 단위가 정의되는데 타임 퀀텀은 일반적으로 10 ~ 100ms의 길이이다. ready queue는 순환 큐로 처리된다.

CPU 스케줄러는 ready queue를 돌면서 각 프로세스에 CPU를 최대 1 타임 퀀텀 (1q)의 시간 간격만큼 할당한다. RR 스케줄링을 구현하기 위해, ready queue를 FIFO 형식으로 처리한다. 새 프로세스가 ready queue에 도착하면 CPU 스케줄러는 ready queue에서 첫 번째 프로세스를 선택하고, 1q 이후 인터럽트할 타이머를 설정하고 프로세스를 디스패치한다. 그러면 두 가지 중 한 가지 일이 일어날 것이다.

1. 프로세스의 CPU 버스트는 1q 사이즈 미만일 수 있는데 이 경우 프로세스 자체가 CPU를 자발적으로 해제한다. 그런 다음 스케줄러는 ready queue에 있는 다음 프로세스로 진행한다.
2. 현재 실행 중인 프로세스의 CPU 버스트가 1q보다 길면 타이머가 꺼지고 운영 체제에 대한 중단이 발생한다. 컨텍스트 스위치가 실행되고 프로세스가 ready queue의 끝에 놓인다. 그런 다음 CPU 스케줄러는 ready queue에 있는 다음 프로세스를 선택한다.

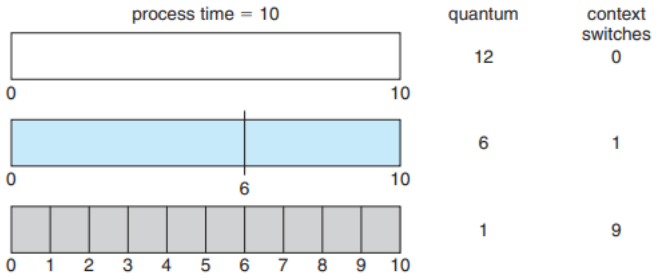
RR 정책에 따른 평균 대기 시간은 종종 길어지곤한다. 아래 예시를 살펴보자.

Process	Burst Time
P_1	24
P_2	3
P_3	3



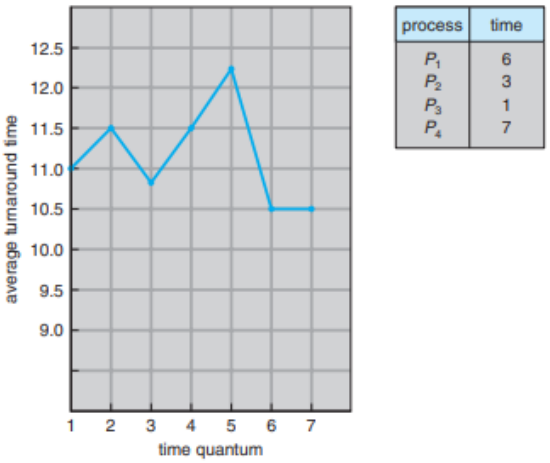
[RR 스케줄링 간트차트]

위의 차트에선 4만큼의 타임 퀀텀을 두었다. P1이 4만큼 실행되고 그 뒤에 P2가 실행된다. 이는 3만큼의 길이를 가지고 있기 때문에 시간7까지 실행되고 다음 프로세스가 실행된다. 다음 프로세스인 P3가 실행되고, P2와 마찬가지로 3만큼의 길이를 가지고 있기 때문에 3이 지난 시간인 10에 실행이 남은 프로세스인 P1이 재실행된다. 이 후 타임퀀텀 길이인 4만큼씩 쪼개져서 실행되며 실행해야할 다른 프로세스가 없기 때문에 계속해서 P1이 종료될때까지 실행된다. 평균 대기 시간을 계산해보자. P1은 6ms, P2는 4ms, P3는 7ms 동안 기다렸으므로 평균 대기 시간은 17/3 = 5.66ms이다. RR 스케줄링 알고리즘에서는, 어떤 프로세스도 연속된 1회 이상의 타임 퀀텀 동안 CPU에 할당되지 않는다. 프로세스의 CPU 버스트가 1q를 초과하면 해당 프로세스가 ready queue에 다시 들어간다. 따라서 RR 스케줄링 알고리즘은 선제적이다. ready queue에 n개의 프로세스가 있고 타임 퀀텀이 q이면 각 프로세스는 최대 q 시간 단위로 CPU 시간의 1/n을 얻는다. 각 과정은 다음 타임 퀀텀까지 (n – 1) × q 시간 단위보다 더 기다려서는 안 된다. RR 알고리즘의 성능은 타임 퀀텀 사이즈에 크게 좌우된다. 극단적으로 타임 퀀텀이 매우 큰 경우, RR 정책은 FCFS 정책과 다를것이 없다. 혹은, 타임 퀀텀이 매우 작을 경우(예: 1ms), RR 접근법은 잦은 컨텍스트 스위칭을 발생시킬 수 있다.



[타임퀀텀 사이즈에 따른 컨텍스트 스위칭 횟수 변화]

예를 들어, 10개의 시간 단위로 이루어진 하나의 프로세스가 있다고 가정한다. 1q의 사이즈가 12시간 단위일 경우 오버헤드 없이 한 번만에 프로세스가 완료된다. 그러나 만약 1q가 6시간 단위라면, 이 프로세스는 2개의 타임 퀀텀을 필요로 하기 때문에 컨텍스트 스위칭이 발생한다. 마지막으로, 타임 퀀텀이 1시간 단위인 경우 9개의 컨텍스트 스위칭이 발생하여 프로세스의 실행 속도가 느려진다. 따라서, 컨텍스트 스위칭과 관련하여 타임 퀀텀은 적절한 크기를 가져야만 한다. 컨텍스트 스위칭 시간이 타임 퀀텀 단위의 약 10%인 경우 CPU 시간의 약 10%가 컨텍스트 스위칭에 사용된다. 실제로, 대부분의 현대 시스템은 10에서 100ms의 타임 퀀텀을 가지고 있다. 컨텍스트 스위칭에 필요한 시간은 일반적으로 10마이크로세컨드 미만이다. 턴어라운드 타임 또한 타임 퀀텀 크기에 따라 달라진다.



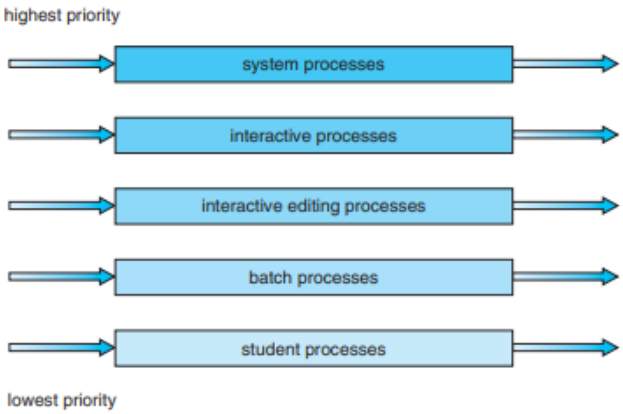
[타임퀀텀 사이즈에 따른 평균 턴어라운드 시간 변화 그래프]

위의 그림에서 알 수 있듯이, 일련의 프로세스의 평균 처리 시간은 타임 퀀텀 사이즈가 증가함에 따라 반드시 개선된다고는 할 수 없다. 일반적으로 대부분의 프로세스가 다음 CPU 버스트를 1 타임 퀀텀 이내로 끝내면 평균 처리 시간이 향상될 수 있다. 타임 퀀텀은 컨텍스트 스위칭 시간과 비교하여 커야 하지만 그렇다고 너무 커서는 안 된다. 타임 퀀텀이 너무 크면 RR 스케줄링은 FCFS 정책으로 전락한다. 프로세스의 80%는 CPU 버스트가 타임 퀀텀보다 짧아야 적정한 수준이라고 할 수 있다. 이 스케줄링은 타임 퀀텀에 따라 응답시간 측면에서 좋은 결과를 낼 수 있는 알고리즘이고 waiting time 측면에서도 나쁘지 않다. 이 방식은 사용자와 교감해야하는 interactive한 작업을 할 때 유용하다. 예를들면 쉘shell이나 브라우저browser같은 프로세스들은 응답이 interactive하게 이뤄지길 원하기 때문에 해당 방식을 이용하면 유용하다.

5.3.5. Multilevel Queue Scheduling

포그라운드 foreground (대화형) 프로세스와 백그라운드 background (배치형) 프로세스는 각각 응답 시간 요구 사항이 다르기 때문에 필요한 스케줄링 방식이 각자 다를 수 있다. 게다가, 포그라운드 프로세스는 백그라운드 프로세스보다 높은 우선 순위(외부적으로 정의된)를 가질 수 있다. 멀티레벨 큐 스케줄링 Multilevel Queue Scheduling 알고리즘은 ready queue를 여러 개의 개별 queue로 나눈다.

프로세스는 일반적으로 메모리 크기, 프로세스 우선 순위 또는 프로세스 유형과 같은 프로세스의 일부 속성을 기반으로 하나의 큐에 영구적으로 할당된다. 각각의 큐에는 고유한 스케줄링 알고리즘이 있다. 예를 들어, 포그라운드 프로세스와 백그라운드 프로세스에는 별도의 큐를 사용할 수 있다. 포그라운드 큐는 RR 알고리즘에 의해 스케줄링되는 동안, 백그라운드 큐는 FCFS 알고리즘에 의해 스케줄링되는것이 가능하다. 또한, 큐들 사이에는 스케줄링이 있어야 하며, 이는 일반적으로 고정 우선 순위 사전 스케줄링 fixed-priority preemptive scheduling으로 구현된다. 예를 들어, 포그라운드 큐는 백그라운드 큐보다 절대적인 우선 순위가 높을 수 있다.



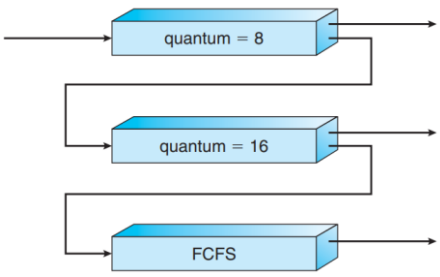
[멀티레벨 큐 스케줄링 시각화]

위 그림의 각 큐는 자신보다 아래에 있는 큐보다 절대적인 우선 순위를 가진다. 예를 들어 시스템 프로세스, 대화형 프로세스 및 대화형 편집 프로세스의 큐가 모두 비어 있지 않으면 배치 큐의 어떤 프로세스도 실행할 수 없다. 배치 프로세스가 실행되는 동안 대화형 편집 프로세스가 ready queue에 진입하면 배치 프로세스가 우선된다. 또 다른 가능성은 큐 간 타임슬라이스를 하는 것이다. 각 큐는 CPU 시간의 일부를 차지하며, 이 시간은 다양한 프로세스 사이에서 스케줄링할 수 있다. 예를 들어 포그라운드-백그라운드 큐의 예시에서 포그라운드 큐는 프로세스 간 RR 스케줄링을 위한 CPU 시간의 80%를 제공할 수 있는 반면, 백그라운드 큐는 FCFS를 기반으로 프로세스에 제공하기 위해 CPU의 20%를 받을 수 있다.

5.3.6. Multilevel Feedback Queue Scheduling

멀티레벨 큐 스케줄링 알고리즘을 사용하면 스케줄링 오버헤드가 낮다는 장점이 있지만 결코 유연하진 않은 반면 멀티레벨 피드백 큐 스케줄링 알고리즘 Multilevel Feedback Queue Scheduling 은 프로세스가 큐 간에 이동할 수 있도록 하여 flexible하다. CPU 버스트의 특성에 따라 프로세스를 분리하는 것이다. 한 프로세스가 CPU 시간을 너무 많이 사용하면 해당 프로세스는 우선 순위가 낮은 큐로 이동되는데 이는 I/O-bound 프로세스와 인터랙티브 프로세스를 고우선순위 큐에 남기게 된다. 또한 낮은 우선순위 큐에서 너무 오래 대기하는 프로세스는 높은 우선순위 큐로 이동할 수 있고 이런 형태의 에이징은 starvation을 예방한다.

예를 들어, 0부터 2까지 번호가 매겨진 3개의 큐를 가진 멀티레벨 피드백 큐 스케줄러를 생각해 보자.



[멀티레벨 피드백 큐]

스케줄러는 먼저 큐 0의 모든 프로세스를 실행한다. 큐 0이 비어 있을 때만 큐 1의 프로세스를 실행한다. 마찬가지로 큐 2의 프로세스는 큐 0과 1이 비어 있는 경우에만 실행된다. 큐 1에 도달하는 프로세스는 큐 2의 프로세스를 우선한다. 큐 1의 프로세스는 큐 0에 도달하는 프로세스에 의해 선행된다. ready queue에 진입하는 프로세스는 큐 0에 놓인다. 큐 0의 프로세스에는 8ms로 설정된 타임 퀀텀이 주어진다. 이 시간 내에 완료되지 않으면 큐 1의 마지막으로 이동한다. 큐 0이 비어 있는 경우, 큐 1의 선두에 있는 프로세스에는 16ms의 타임 퀀텀이 주어진다. 그 시간동안 완료되지 못하면 큐 2에 들어간다. 큐 2의 프로세스는 FCFS 기반으로 실행되지만 큐 0 및 1이 비어 있는 경우에만 실행된다. 이 스케줄링 알고리즘은 CPU 버스트가 8ms 이하인 모든 프로세스에 가장 높은 우선 순위를 부여한다. 이러한 프로세스는 CPU를 빠르게 가져오고 CPU 버스트를 완료하며 다음 I/O 버스트로 전환된다. 8에서 24ms 사이의 시간이 필요한 프로세스도 빠르게 처리되지만, 더 짧은 프로세스보다 우선순위는 낮다. 긴 프로세스는 자동으로 큐 2에 들어가고 큐 0 및 1에서 남은 CPU 사이클과 함께 FCFS 순서로 처리된다.

일반적으로 멀티레벨 피드백 큐 스케줄러는 다음과 같은 매개 변수로 정의된다.

- 큐의 수
- 각 큐의 스케줄링 알고리즘
- 프로세스를 높은 우선 순위 큐로 업그레이드할 시기를 결정하는 데 사용되는 방법
- 프로세스를 낮은 우선 순위 큐로 강등할 시기를 결정하는 데 사용되는 방법
- 프로세스가 필요에 의해 들어갈 큐를 결정하는 데 사용되는 방법

멀티레벨 피드백 큐 스케줄러의 정의는 가장 일반적인 CPU 스케줄링 알고리즘으로 만든다. 어떤 특정한 시스템과 맞도록 구성할 수 있다. 그러나 최적의 스케줄러를 정의하려면 모든 매개 변수에 대한 값을 선택하는 방법이 필요하기 때문에 이 알고리즘은 가장 복잡한 알고리즘이기도 하다.

5.4. Multiple-Processor Scheduling

여러 CPU를 사용할 수 있으면 로드 공유가 가능해지지만 스케줄링 문제도 그만큼 복잡해진다. 멀티프로세서 스케줄링에 대해 알아보려한다.

5.4.1. Approaches to Multiple-Processor Scheduling

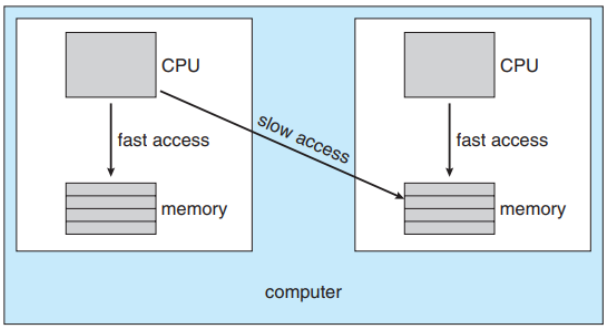
멀티프로세서 시스템의 첫 번째 방식. 모든 스케줄링 결정, I/O 처리 및 기타 시스템 활동을 단일 프로세서에 의해 처리한다. 다른 프로세서는 오직 사용자 코드만 실행한다. 이러한 비대칭 멀티프로세싱은 하나의 프로세서만이 시스템 데이터 구조에 액세스하기 때문에 데이터 공유의 필요성을 줄여주어서 간단하다고 말할 수 있다.

두 번째 방식은 각 프로세서가 자체 스케줄링하는 대칭symmetric 다중 처리(SMP)를 사용한다. 모든 프로세스가 공통 ready queue에 있거나 각 프로세서가 준비 프로세스의 전용 큐를 가질 수

있다. 그럼에도 불구하고 스케줄링은 각 프로세서에 대한 스케줄러가 준비 대기열을 검사하고 실행할 프로세스를 선택하도록 하여 진행된다. 공통 데이터 구조에 액세스하고 업데이트하려는 프로세서가 여러 개 있는 경우 스케줄러는 신중하게 프로그래밍되어야 한다. 두 개의 분리된 프로세서가 동일한 프로세스를 예약하지 않으며 대기열에서 손실되지 않도록 해야 한다. 거의 모든 최신 운영 체제는 SMP를 지원한다.

5.4.2. Processor Affinity

프로세스가 특정 프로세서에서 실행되었을 때 캐시 메모리는 어떻게 되는가? 프로세스에서 가장 최근에 액세스한 데이터가 프로세서의 캐시를 채운다. 결과적으로 프로세스에 의한 연속적인 메모리 액세스는 종종 캐시 메모리에서 충족된다. 만일 프로세스가 다른 프로세서로 마이그레이션된다면 그 경우, 캐시 메모리의 내용은 첫 번째 프로세서에 대해 무효화되어야 하며 두 번째 프로세서에 대한 캐시는 다시 채워져야 한다. 캐시를 무효화하고 다시 채우는 높은 비용 때문에, 대부분의 SMP 시스템은 한 프로세서에서 다른 프로세서로 프로세스를 마이그레이션하는 것을 피하고 대신 동일한 프로세서에서 프로세스를 계속 실행하려고 시도하는데 이를 프로세서 선호도라고 한다. 즉, 프로세스가 현재 실행 중인 프로세서에 대한 선호도를 가진다. 프로세서 선호도는 여러 가지 형태를 취한다. 운영 체제가 프로세스를 동일한 프로세서에서 계속 실행하도록 시도하는 정책을 가지고 있을 때, 우리는 소프트 어피니티 Affinity 라고 불리는 상황이 발생한다. 여기서 운영 체제는 단일 프로세서에 프로세스를 유지하려고 시도하지만 프로세서를 마이그레이션하는 프로세스가 가능하다. 이와는 대조적으로, 일부 시스템은 하드 어피니티를 지원하는 시스템 호출을 제공하므로 프로세스가 실행될 수 있는 프로세서의 하위 집합을 지정할 수 있다.



[NUMA와 CPU스케줄링]

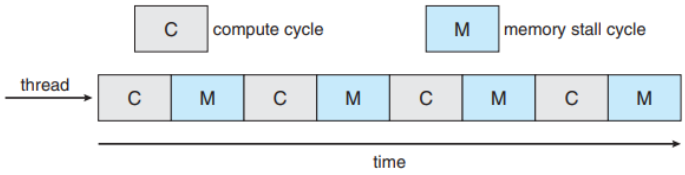
시스템의 메인 메모리 아키텍처는 프로세서 친화성 문제에 영향을 미칠 수 있다. 위의 그림은 CPU가 다른 부분보다 메인 메모리의 일부 부분에 더 빠르게 접근할 수 있는 NUMA (Non-uniform memory access)를 특징으로 하는 아키텍처를 보여준다. 일반적으로 CPU와 메모리 보드가 결합된 시스템에서 발생한다. 보드의 CPU는 시스템의 다른 보드의 메모리에 액세스하는 것보다 더 빨리 해당 보드의 메모리에 액세스할 수 있다. 운영 체제의 CPU 스케줄러와 메모리 배치 알고리즘이 함께 작동하면 특정 CPU에 친화력이 할당된 프로세스가 해당 CPU가 있는 보드에 메모리를 할당할 수 있다.

5.4.3. Load Balancing

로드 밸런싱은 SMP 시스템의 모든 프로세서에 작업 부하가 균등하게 분산되도록 시도한다. 일반적으로 로드 밸런싱은 각 프로세서가 실행할 수 있는 프로세스의 전용 큐를 가진 시스템에서만 필요한다. 공통 실행 대기열이 있는 시스템에서는 프로세서가 유휴 상태가 되면 실행 가능한 프로세스를 공통 실행 대기열에서 즉시 추출하기 때문에 로드 밸런싱이 필요하지 않은 경우가 많으나 SMP를 지원하는 대부분의 최신 운영 체제에서는 각 프로세서가 적격 프로세스의 개인 대기열을 가지고 있다는 점도 유의해야 한다. 로드 밸런싱에는 푸시 마이그레이션과 풀 마이그레이션이라는 두 가지 일반적인 방법이 있다. 푸시 마이그레이션을 사용하면 특정 작업이 각 프로세서의 로드를 주기적으로 검사하고, 불균형을 발견하면 오버로드된 프로세서에서 유휴 또는 사용량이 적은 프로세서로 프로세스를 이동하여 로드를 균등하게 분산시킨다. 풀 마이그레이션은 유휴 프로세서가 사용 중인 프로세서에서 대기 중인 태스크를 가져올 때 발생한다. 푸시 및 풀 마이그레이션은 상호 배타적일 필요가 없으며 실제로 로드 밸런싱 시스템에서 병렬로 구현되는 경우가 많다. 로드 밸런싱은 종종 프로세서 선호도의 이점을 상쇄한다. 즉, 동일한 프로세서에서 프로세스를 실행함으로써 얻을 수 있는 이점은 프로세스가 프로세서의 캐시 메모리에 있는 데이터를 활용할 수 있다는 것이다. 프로세스를 한 프로세서에서 다른 프로세서로 끌어당기거나 밀어넣으면 이러한 이점이 제거된다. 따라서 일부 시스템에서 유휴 프로세서는 항상 유휴 프로세서가 아닌 프로세서에서 프로세스를 끌어온다. 다른 시스템에서는 불균형이 특정 임계값을 초과하는 경우에만 프로세스가 이동된다.

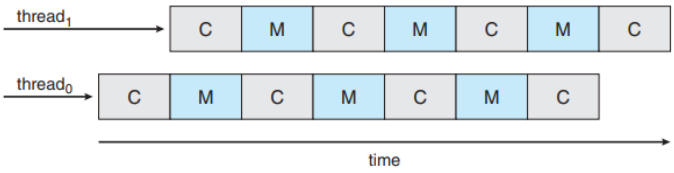
5.4.4. Multicore Processors

전부터 SMP 시스템은 여러 개의 물리적 프로세서를 제공하여 여러 개의 스레드를 동시에 실행할 수 있게 해왔으나 최근 컴퓨터 하드웨어의 관행은 다중 프로세서 코어를 동일한 물리적 칩에 배치하여 다중 코어 프로세서를 만드는 것이다. 각 코어는 아키텍처 상태를 유지하므로 운영 체제에서는 별도의 물리적 프로세서로 보여준다. 멀티코어 프로세서를 사용하는 SMP 시스템은 각 프로세서가 자체 물리적 칩을 가지고 있는 시스템보다 더 빠르고 더 적은 전력을 소비한다. 멀티코어 프로세서는 스케줄링 문제를 복잡하게 할 수 있다.



[메모리 스톨]

프로세서가 메모리에 액세스할 때 데이터를 사용할 수 있을 때까지 상당한 시간을 소비하는 상황을 메모리 스톨이라 하는데 이는 캐시 메모리에 없는 데이터에 액세스하는 것과 같은 다양한 이유로 발생할 수 있다. 위의 그림과 같이 프로세서는 최대 50%의 시간을 메모리에서 데이터를 사용할 수 있도록 대기하는 데 소비할 수 있는데, 이를 개선하기 위해 최근의 많은 하드웨어 설계에서는 멀티 스레드 프로세서 코어를 구현하고 있으며, 각 코어에 두 개 이상의 하드웨어 스레드가 할당되어 있다. 이렇게 하면 메모리를 기다리는 동안 한 스레드가 멈추면 코어가 다른 스레드로 전환될 수 있다.



[멀티스레디드 멀티코어 시스템]

위의 그림은 스레드 0의 실행과 스레드 1의 실행이 인터리빙된 듀얼 스레드 프로세서 코어를 보여주는것인데 운영 체제의 관점에서 각 하드웨어 스레드는 소프트웨어 스레드를 실행할 수 있는 논리 프로세서로 나타난다. 따라서 듀얼 스레드 듀얼 코어 시스템에서 4개의 논리 프로세서가 운영 체제에 제공된다.

일반적으로 프로세싱 코어를 멀티스레딩할때 거친 입자의 멀티스레딩coarse grained multithreading과 미세한 입자의 멀티스레딩fine-grained multithreading 두 가지의 방법을 사용한다. coarse grained multithreading으로 스레드는 메모리 스톨과 같은 긴 지연 이벤트가 발생할 때까지 프로세서에서 실행되고 지연 시간이 긴 이벤트로 인한 지연으로 인해 프로세서가 실행을 시작하려면 다른 스레드로 전환해야 한다. 그러나 명령어 파이프라인을 플러시해야 다른 스레드가 프로세서 코어에서 실행을 시작할 수 있기 때문에 스레드 간 전환 비용이 많이 든다. 이 새 스레드가 실행을 시작하면 파이프라인에 지시사항을 채우기 시작한다. 세분화된 멀티 스레드 스위치는 훨씬 더 세분화된 레벨로 스레드 간에 전환된다. 그러나 세밀한 시스템의 아키텍처 설계에는 스레드 스위칭을 위한 논리가 포함되어 결과적으로, 스레드 간 전환 비용이 적다. 멀티스레드 멀티코어 프로세서는 실제로 두 가지 다른 레벨의 스케줄링이 필요하다. 한 레벨에서 운영 체제가 각 하드웨어 스레드(논리 프로세서)에서 실행할 소프트웨어 스레드를 선택할 때 내려야 하는 스케줄링 결정이 있다. 이 레벨의 스케줄링을 위해 운영 체제는 임의의 스케줄링 알고리즘을 선택할 수 있다. 두 번째 레벨의 스케줄링에서는 각 코어가 실행할 하드웨어 스레드를 결정하는 방법을 지정한다.

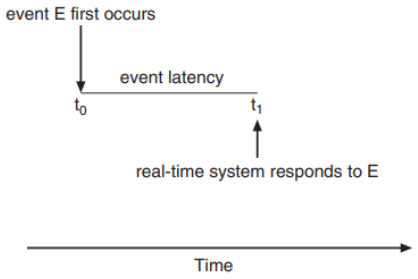
5.5. Real-Time CPU Scheduling

실시간 운영 체제에 대한 CPU 스케줄링에는 특별한 문제가 수반된다. 소프트 실시간 시스템은 중요한 실시간 프로세스가 언제 예약될지에 대한 보장을 제공하지 않으며 프로세스가 덜 중요한 프로세스보다 우선시된다는 것만 보장한다. 하드 실시간 시스템은 더 엄격한 요구 사항을 가진다. 작업은 데드라인까지 서비스를 받아야 하며 데드라인이 만료된 후의 서비스는 서비스가 없는 것과 같다. 이 섹션에서는 관련된 몇 가지 문제를 살펴본다.

5.5.1. Minimizing Latency

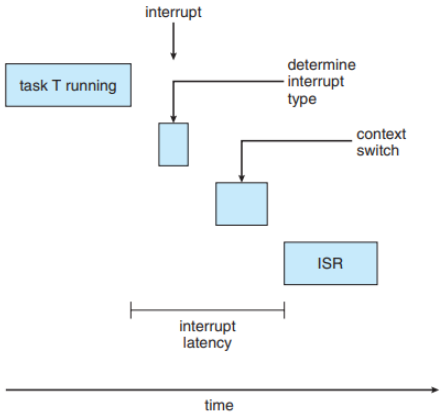
실시간 시스템의 이벤트 중심 특성을 고려해보자. 시스템은 일반적으로 실시간으로 이벤트가 발생하기를 기다린다. 이벤트는 소프트웨어(타이머가 만료되는 경우) 또는 하드웨어(원격 제어

차량이 장애물에 접근하고 있음을 감지하는 경우)에서 발생하는데 이벤트가 발생하면 시스템은 가능한 한 신속하게 응답하고 서비스를 제공해야 한다.



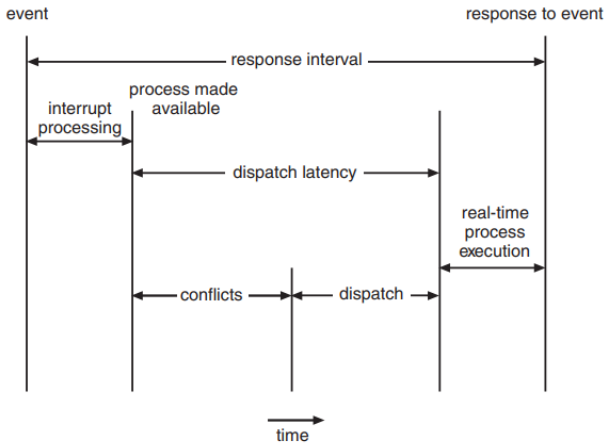
[이벤트 latency]

이벤트 지연 시간event latency은 이벤트가 발생한 시점부터 서비스가 제공될 때까지의 시간인데 일반적으로 이벤트마다 지연 시간 요구 사항이 다르며 두 가지 유형의 대기 시간이 실시간 시스템의 성능에 영향을 미친다.



[인터럽트 latency]

1. 인터럽트 지연 시간 – CPU에 인터럽트가 도착한 후 인터럽트를 처리하는 루틴이 시작될 때까지의 기간이다. 인터럽트가 발생하면 운영 체제는 먼저 실행 중인 명령을 완료하고 발생한 인터럽트의 유형을 결정해야 한다. 그런 다음 특정 인터럽트 서비스 루틴(ISR)을 사용하여 인터럽트를 처리하기 전에 현재 프로세스의 상태를 저장해야 하는데 이러한 작업을 수행하는 데 필요한 총 시간은 인터럽트 지연 시간이다. 실제로, 하드 실시간 시스템의 경우, 인터럽트 지연 시간을 단순히 최소화해서는 안 되며 실시간 작업이 즉각적인 주의를 받도록 하기 위해 인터럽트 지연을 최소화하는 것이 중요하다. 인터럽트 지연에 기여하는 한 가지 중요한 요소는 커널 데이터 구조가 업데이트되는 동안 인터럽트가 비활성화될 수 있는 시간의 양이다. 실시간 운영 체제는 매우 짧은 시간 동안만 인터럽트를 비활성화해야 한다.



[디스패치 latency]

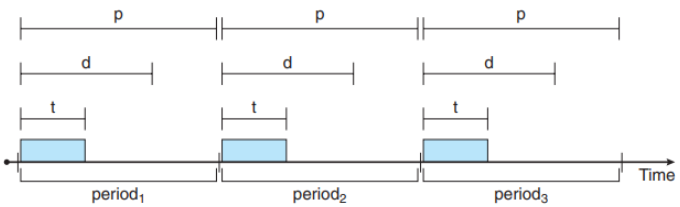
2. 디스패치 지연 시간 – 스케줄링 디스패처가 한 프로세스를 중지하고 다른 프로세스를 시작하는 데 필요한 시간을 디스패치 지연 시간이라고 한다. CPU에 즉시 액세스할 수 있는 실시간 작업을 제공하려면 실시간 운영 체제에서도 이러한 지연 시간을 최소화해야 한다. 디스패치 지연 시간을 낮게 유지하는 가장 효과적인 기술은 선제적 커널을 제공하는 것이다.

   디스패치 지연 시간의 충돌 단계에는 두 가지 구성 요소가 있다.
   a. 커널에서 실행 중인 프로세스의 preemption
   b. 저우선순위 프로세스에 의한 고우선순위 프로세스에 필요한 리소스의 릴리즈

5.5.2. Priority-Based Scheduling

실시간 운영 체제의 가장 중요한 특징은 CPU가 필요한 즉시 실시간 프로세스에 즉시 반응하는 것이다. 따라서 실시간 운영 체제의 스케줄러는 우선 순위 기반 알고리즘을 선점하여 지원해야 한다. 우선순위 기반 스케줄링 알고리즘은 각 프로세스의 중요도에 따라 우선순위를 부여한다. 스케줄러가 preempt도 지원하는 경우, 더 높은 우선순위의 프로세스를 실행할 수 있게 되면 CPU에서 현재 실행 중인 프로세스가 preempt된다.

스케줄링할 프로세스의 특정 특성을 정의해보자면 일단, 과정은 주기적인 것으로 간주된다. 즉, 일정한 간격(주기)으로 CPU가 필요하다. 주기적인 프로세스가 CPU를 획득하면, 그것은 고정된 처리 시간 t, CPU에 의해 서비스되어야 하는 마감 시간, 그리고 p 기간을 갖는다. 상기 처리 시간, 마감 시간 및 기간의 관계는 $0 \le t \le d \le p$로 표현될 수 있다. 주기적인 작업의 비율은 1/p이다. 아래 그림은 시간 경과에 따른 주기적 프로세스의 실행을 보여준다.



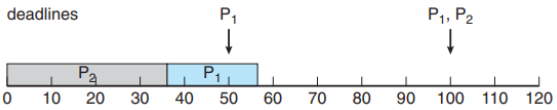
[Periodic task]

스케줄러는 이러한 특성을 활용하고 프로세스의 데드라인 또는 요율 요건에 따라 우선순위를 할당할 수 있다. 이러한 형태의 스케줄링에서 특이한 점은 프로세스가 스케줄러에 데드라인 요구사항을 알려야 한다는것이다. 그런 다음, 승인 제어 알고리즘으로 알려진 기술을 사용하여 스케줄러는 프로세스를 인정하여 프로세스가 제 시간에 완료될 것을 보장하거나, 데드라인까지 작업이 수행될 것이라고 보장할 수 없는 경우 요청을 불가능으로 거부한다.

5.5.3. Rate-Monotonic Scheduling

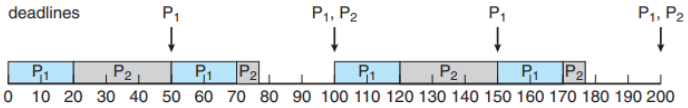
Rate-monotonic scheduling 알고리즘은 preemption이 있는 정적 우선 순위 정책 static priority policy을 사용하여 주기적인 작업을 스케줄링한다. 저우선순위 프로세스가 실행 중이고 고우선순위 프로세스를 실행할 수 있게 되면 저우선순위 프로세스가 우선된다. 시스템에 들어가면, 각각의 주기적인 작업은 그 주기에 반비례하여 우선순위가 할당된다. 기간이 짧을수록 높아지고, 길수록 낮아진다. 이 정책의 근거는 CPU가 더 자주 필요한 작업에 더 높은 우선순위를 할당하는 것이다. 또한, Rate-monotonic scheduling에서는 주기적 프로세스의 처리 시간이 각 CPU 버스트에 대해 동일하다고 가정한다. 즉, 프로세스가 CPU를 획득할 때마다 CPU 버스트의 지속 시간은 동일하다.

예를 들어, P_1과 P_2의 주기는 각각 $p_1 = 50$과 $p_2 = 100$이다. 처리 시간은 P_1의 경우 $t_1 = 20$, P_2의 경우 $t_2 = 35$이다. 각 프로세스의 데드라인은 다음 주기가 시작되기 전까지이다. 우리는 먼저 각자가 데드라인을 준수할 수 있도록 이러한 작업 일정을 잡는 것이 가능한지 자문해 보아야 한다. 프로세스 P_i의 CPU 사용률을 주기(t_i/p_i)에 대한 버스트의 비율로 측정하면 P_1의 CPU 사용률은 20/50 = 0.40이고 P_2의 CPU 사용률은 35/100 = 0.35이며, 총 CPU 사용률은 75%이다. 따라서 둘 다 데드라인을 준수하고 CPU에 사용 가능한 주기를 남겨두는 방식으로 이러한 작업을 스케줄링할 수 있을 것으로 보인다.



[P2가 P1보다 중요도가 높을때의 Rate-monotonic 스케줄링]

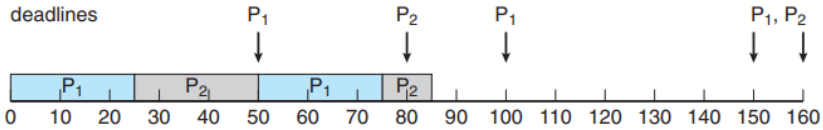
위와 같이 P_2에 P_1보다 더 높은 우선 순위를 할당한다고 가정하자. P_2는 먼저 실행을 시작하고 35에 완료한다. 이 시점에서 P_1은 시작되며, 55에 CPU 버스트를 완료하는데 P_1의 첫 번째 마감은 50이었기 때문에 스케줄러는 P_1의 마감을 놓쳤다.



[P1가 P2보다 중요도가 높을때의 Rate-monotonic 스케줄링]

P_1의 주기가 P_2의 주기에 비해 짧기 때문에 P_1을 P_2보다 더 높은 우선 순위를 할당하는 Rate-monotonic scheduling을 사용하는 예를 보자. P_1은 먼저 시작하고 시간 20에 CPU 버스트를 완료하여 첫 번째 데드라인을 준수한다. P,는 이 지점에서 실행을 시작하여 시간 50까지 실행된다. 이때 P_1에 의해 선점되지만 CPU 버스트에 아직 5ms가 남아 있다. P_1은 70에 CPU 버스트를 완료하고, 그 시점에서 스케줄러는 P_2를 재개한다. P_2는 CPU 버스트를 시간 75에 완료하고 첫 번째 데드라인을 준수한다. 시스템은 P_1이 다시 스케줄링되는 시간인 100까지 유휴 상태이다. Rate-monotonic scheduling은 이 알고리즘에 의해 일련의 프로세스를 스케줄링할 수 없는 경우 최적이라고 간주된다.

Rate-monotonic algorithm을 사용하여 스케줄링할 수 없는 프로세스 집합을 살펴보자. 프로세스 P_1의 주기가 $p_1 = 50$이고 CPU 버스트가 $t_1 = 25$, P_2은 $p_2 = 80$이고 $t_2 = 35$ 일 때, Rate-monotonic scheduling은 프로세스 P_1에 더 높은 우선순위를 부여한다. 총 CPU 사용률은 (25/50)+ (35/80) = 0.94이므로 두 프로세스를 스케줄링할 수 있고 6%의 사용 가능한 시간이 남아 있으므로 이는 논리적으로 보인다.



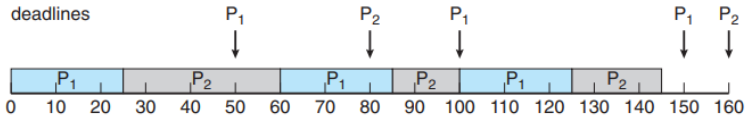
[rate-monotonic 스케줄링 중 데드라인을 놓친 상황]

Rate-monotonic scheduling에는 한계가 있다. 예를 들어 처음에 P_1은 시간 25에 CPU 버스트를 완료할 때까지 실행된다. 그런 다음 프로세스 P_2가 실행 시작하여 P_1에 의해 뺏기는 순간까지 실행된다. 이 시점에서 P_2의 CPU 버스트에는 아직 10ms가 남아 있다. 프로세스 P_1은 시간 75까지 실행되며, 따라서 P_2는 시간 80에 CPU 버스트를 완료하기 위한 데드라인을 놓친다. CPU 활용도가 제한적이며 CPU 리소스를 최대화하는 것이 항상 완전히 가능한 것이 아니라는 한계를 보여준다. N개의 프로세스 스케줄링에 대한 최악의 CPU 사용률은 $N(2^{1/N} - 1)$이다.

시스템에 한 개, 두 개, 무한대의 프로세스가 있다면 CPU 사용률은 각 100%, 83%, 69%이다. 위의 첫 번째, 두번째 예시의 CPU 사용률을 합치면 75%가 되므로 Rate-monotonic scheduling 알고리즘은 데드라인을 준수할 수 있도록 해당 프로세스를 스케줄링할 수 있다. 그러나 마지막 예시의 두 프로세스 CPU 활용률은 약 94%이므로 Rate-monotonic scheduling으로는 데드라인을 준수할 수 있다고 보장할 수 없다.

5.5.4. Earliest-Deadline-First Scheduling

EDF 스케줄링의 설명을 위해, 이전 섹션의 데드라인을 지키지 못한 마지막 예시의 프로세스를 다시 스케줄링한다. P_1의 값이 $p_1 = 50$, $t_1 = 25$이며 P_2의 값이 $p_2 = 80$, $t_2 = 35$ 였었다.



[EDF 스케줄링]

P_1은 데드라인이 제일 빨라서 초기 우선순위가 P_2보다 높다. P_2는 P_1의 CPU 버스트 끝에서 실행되기 시작한다. Rate-monotonic scheduling이었다면 P_1은 50에 P_2를 preempt할 수 있지만 EDF 스케줄링에선 P_2가 계속 실행될 수 있다. P_2는 다음 데드라인(80)이 P_1(100)보다 빠르기 때문에 P_1보다 더 높은 우선순위를 갖는다. 따라서 P_1과 P_2는 모두 첫 번째 데드라인을 준수한다. P_1은 60에 다시 실행을 시작하고 85에 두 번째 CPU 버스트를 완료하며 두 번째 데드라인도 충족한다. P_2는 이 시점에서 실행되기 시작하지만, 100에서 다음 주기가 시작될 때 P_1이 P_2보다 더 이른 데드라인을 가지고 있기 때문에 P_1이 선점한다. 125에 P_1은 CPU 버스트를 완료하고 P_2는 실행을 재개하여 145에 종료하고 데드라인도 준수한다. 시스템은 P_1이 다시 실행되도록 스케줄링된 150까지 유휴 상태이다. Rate-monotonic algorithm과 달리 EDF 스케줄링은 프로세스가 주기적일 periodic 필요도, 버스트당 일정한 CPU 시간을 요구 할 필요도 없다. 유일한 요구 사항은 프로세스가 실행 가능해질 때 스케줄러에 데드라인을 알리는 것이다. EDF 스케줄링은 이론적으로는 각 프로세스가 데드라인 요구 사항을 충족하고 CPU 활용률이 100%가 되도록 프로세스를 스케줄링할 수 있기 때문에 최적인것처럼 보이나 실제로는 프로세스 간의 컨텍스트 스위칭과 인터럽트 처리 비용 때문에 이 수준의 CPU 활용도를 달성하는 것이 불가능하다.

5.5.5. Proportional Share Scheduling

Proportional Share Scheduling 은 모든 애플리케이션에 *T* 를 할당해 운영한다. 응용 프로그램은 *N*개의 타임 쉐어링을 수신할 수 있으므로 응용 프로그램이 총 프로세서 시간의 *N/T* 를 가질 수 있다.

예를 들어, 총 *T* = 100 를 A, B, C 세 프로세스들에 각 A 50, B 15, C 20를 배정했다고 가정하자. 이 방식은 A가 전체 프로세서 시간의 50%를, B가 15%를, C가 20%를 갖도록 보장한다. Proportional Share Scheduling은 애플리케이션이 할당된 시간 몫을 받도록 보장하기 위해 승인 제어 정책과 함께 작업해야 한다. 승인 제어 정책은 충분한 공유를 사용할 수 있는 경우에만 특정 수의 공유를 요청하는 클라이언트를 허용한다. 현재 예제에서는 총 100 중 50 + 15 + 20 = 85를 할당했기 때문에 새로운 프로세스인 D가 30을 요청하면, 승인 컨트롤러는 D의 시스템 진입을 거부한다.

5.5.6. POSIX Real-Time Scheduling

기본적으로 POSIX는 실시간 컴퓨팅을 위한 API를 제공하며 실시간 스레드에 대한 두 가지 스케줄링 클래스를 정의한다

- SCHED_FIFO - SCHED_FIFO는 FIFO 대기열을 사용하여 선착순 정책에 따라 스레드를 스케줄링한다. 그러나 동일한 우선 순위의 스레드 간에 슬라이싱할 시간이

없기 때문에 FIFO 대기열의 맨 앞에 있는 최고우선순위 실시간 스레드는 종료되거나 차단될 때까지 CPU에 부여된다.

- SCHED_RR - SCHED_RR은 라운드 로빈 정책을 사용한다. 동일한 우선 순위의 스레드 간에 시간 슬라이싱을 제공한다는 점을 제외하면 SCHED_FIFO와 유사하다.

POSIX API는 스케줄링 정책을 가져오고 설정하기 위한 두 가지 함수를 지정한다.

- pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)
  (속성 집합에 대한 포인터, 현재 예약 정책으로 설정된 정수에 대한 포인터)
- pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)
  (속성 집합에 대한 포인터, 정수 값(SCHED_FIFO, SCHED_RR))

두 함수 모두 오류가 발생하면 0이 아닌 값을 반환한다.

Abraham Silberschatz, Peter B Galvin and Greg Gagne, *Operating system concepts* 9th ed. (2012, Wiley)