

내용

BSD 유닉스	3
운영체제 개요	4
운영체제 구조	4
운영체제의 발전	4
프로세스의 모든 것	5
Process	5
Process Image	5
Process State Transition	7
PCB	7
Process Scheduling 목적	8
Long-term Scheduler	8
Short-term Scheduler	8
Medium-term Scheduler	9
Context-switching	9
CPU Dispatcher	10
프로세스 생성과 종료	11
Fork	11
Exec	11
Wait	12
고아(Orphan) 프로세스	12
좀비(Zombie) 프로세스	12
프로세스 종료	13
프로세스 간 통신	13
IPC : Inter-Process Communication	13
IPC Model	13
1. Shared Memory Model	13
2. Message Passing Model	14

메시지 전송 모델의 장점	14
메시지 전송 모델의 단점	14
Rendezvous	15
Buffering	15
3. Socket	16
4. RPC : Remote Procedure Calls	16
5. Pipe	17
Thread	18
그럼 다중 프로세스와 다중 스레드의 차이점은 무엇일까	18
멀티 스레드 장점	19
멀티 스레드 모델	20
스레딩 방식	21
Implicit Threading	22
Thread 관련 문제	23
fork()와 exec() 시스템호출의 의미	23
Thread Cancellation(스레드 실행 취소)	23

BSD 유닉스

BSD : Berkeley Software Distribution

유닉스 개발 - 1973년 SOSP컨퍼런스에서 벨 연구소 켄 톰슨, 데니스 리치가 발표

SOSP : Symposium of Operationg Systems Principles

유닉스 - C언어로 개발되었으며 여러 사용자가 동시에 사용 할 수 있는 대화형 운영체제

적용 기종 - PDP-11

Vi 에디터 - 빌조이가 유닉스에서 비주얼 편집 모드가 되도록 개발함

3BSD - 32비트 VAX에 맞게 2BSD를 이식 시킨 버전

미국방성에서 운영체제를 네트워크 상에서 운용하기 위해 버클리대학교에 연구/개발을 맡김, 밥 패브릭 교수가 CSRG(Computer Systems Research Group)을 결성하여 제작하기로 함.

구성인원 : 빌 조이 등..

5BSD를 출시하려 하였으나 AT&T가 시스템V와 혼동할 수 있다고 판단하여 이름을 변경할 것을 요구 그래서 4.1BSD로 출시

BSD에 TCP/IP를 추가한 버전인 4.2BSD가 출시됨

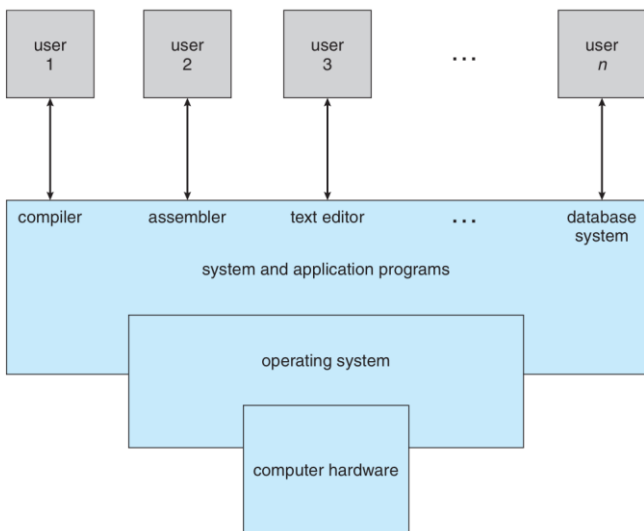
운영체제 개요

운영체제 구조

운영체제란 컴퓨터 자원을 관리하는 프로그램

운영체제는 기본적인 응용 프로그램을 제공하며 컴퓨터 하드웨어와 컴퓨터 사용자 사이의 중계 역할을 수행한다. 어떤 운영체제는 편리하거나 효율적으로 설계되고 다른 건 이 둘을 조합하기도 한다.

컴퓨터 시스템은 하드웨어, 운영체제, 응용 프로그램, 사용자 이 4가지로 나타낼 수 있다.



CPU, 메모리, I/O장치로 구성된 하드웨어는 시스템에 기본적인 연산 자원을 제공한다. 워드, 웹 브라우저 같은 응용 프로그램은 사용자의 연산 문제를 해결하기 위해 사용 되는 이러한 자원(하드웨어)들의 연산 방법을 정의한다.

Figure 1.1 Abstract view of the components of a computer system.

운영체제의 관점은 사용자와 시스템으로 나뉜다. 사용자 관점에서 운영체제는 한 명의 사용자가 하드웨어 자원을 독점하도록 설계된다. 이러한 이유는 사용자가 하는 작업을 최대화하기 위함이다. 즉 사용의 편리함과 자원의 활용을 추구한다. 시스템(컴퓨터)의 관점에서 운영체제는 자원을 할당하고 관리하는 것이다. 자원 할당은 특히 같은 서버에서 많은 사용자가 접속하는 환경에서 중요하다. 즉 제어 프로그램을 의미한다.

정리

사용자 관점 - 사용/습득 용이성, 신뢰성, 안전성, 성능

시스템 관점 - 설계/구현/유지보수 용이성, 신뢰성, 오류로 부터 자유, 효율성

운영체제의 발전

1. 직렬처리 (Serial Processing) - 운영체제 없이 프로그래머가 직접 H/W를 제어
2. 일괄처리 (Batch Processing) - 들어온 Job을 큐에 저장하고 Job을 순차적으로 수행한다.. 입출력이 발생

하는 경우 완료 될 때 까지 CPU는 기다린다.

3. 다중 프로그래밍 (Multiprogramming) – 동시에 여러개의 프로세스가 돌아가는 것 처럼 보이는 환경이다.
 - A. 입출력 별도의 I/O Processor에 의해 실행되므로 그동안 CPU는 다른 작업을 수행 할 수 있음, 입출력이 완료되면 CPU에 Interrupt 신호를 전송하여 작업을 재행
 - B. 입출력 작업과 CPU 작업을 동시에 수행하므로 CPU 효율성 증가
4. 시분할 시스템 (Time Sharing System) – 다수의 사용자(프로그램)가 CPU time을 공유하며 동시에 실행되는 것 처럼 보임, time slice 경과시 interrupt가 발생되고 CPU scheduling(p STS)이 수행됨

프로세스의 모든 것

Process

프로세스의 정의는 실행중인 프로그램을 의미한다. 프로세스는 프로그램의 파일안에 있는 명령어 코드를 수행한다. 또한 프로세스는 시스템의 자원을 시분할로 공유하므로 PC(Program Counter)의 값과 프로세서의 레지스터 내용을 가지고 있다. 프로세스 마다 별도의 자원을 관리하기 위해 임시 데이터(함수의 매개변수, 반환 주소, 지역 변수)와 전역변수를 가지고 있는 데이터 섹션을 포함한다. 프로세스 실행 동안 동적으로 메모리를 할당할 수도 있다. 이를 힙이라 한다. 즉 프로세스는 메모리에 다음과 같이 할당된다.

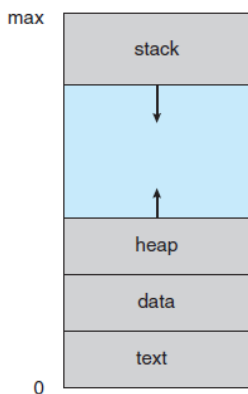


Figure 3.1 Process in memory.

과거 컴퓨터는 한 번에 하나의 프로그램만 실행할 수 있게 했다. 이 프로그램은 시스템을 완전히 제어하고 모든 시스템의 자원을 접근할 수 있었다. 반면 현대 컴퓨터는 여러 개의 프로그램을 메모리에 올려 동시에 실행할 수 있다. 이를 위해 각각의 프로그램마다 강력한 제어와 구획화가 필요했다. 이러한 필요에서 실행 중인 프로그램인 프로세스라는 개념이 나왔다. 프로세스는 현대의 시분할 시스템의 작업단위이다. 프로세스는 시스템 코드를 실행하는 운영체제 프로세스와 사용자 프로그램 코드를 실행하는 사용자 프로세스로 구성된다. 이들 프로세스를 프로세서에서 일정 시간 마다 처리하면서 컴퓨터를 더 생산적으로 만들게 되었다.

Process Image

현대 컴퓨터 시스템은 사용자의 생산성을 높이기 위해 시분할을 통한 다중 프로세스가 가능하다. 프로세서가 시분할로 프로세스를 처리하기 위해서는 현재 프로세스와 다음 프로세스의 정보 저장과 불러오기가 필요하다. 이를 위해 프로세스 이미지를 사용한다. 프로세스 이미지는 어느 시점의 프로세스 정보를 담고 있다. 그 정보는 사용자 프로그램, 사용자 데이터 (함수의 매개변수, 반환 주소, 지역 변수, 동적 할당 메모리 등), PCB(Process Control Block)로 구성된다.

User Program
User Data: stack, heap, data.
PCB <i>(context or state of the process)</i>

} User Address Space
(사용자 주소 공간)

- 프로세스 실행 관련 정보:
(Process Control Block)
- 프로세스 식별자
 - CPU 스케줄링 정보
 - 자원 할당 정보
 - 등등

Process State Transition

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

프로세스의 상태는 해당 프로세스의 현재 활동에 의해 정의된다. 프로세스의 상태는 다음과 같다.

New : 해당 프로그램의 객체(프로세스)를 생성 중이다.

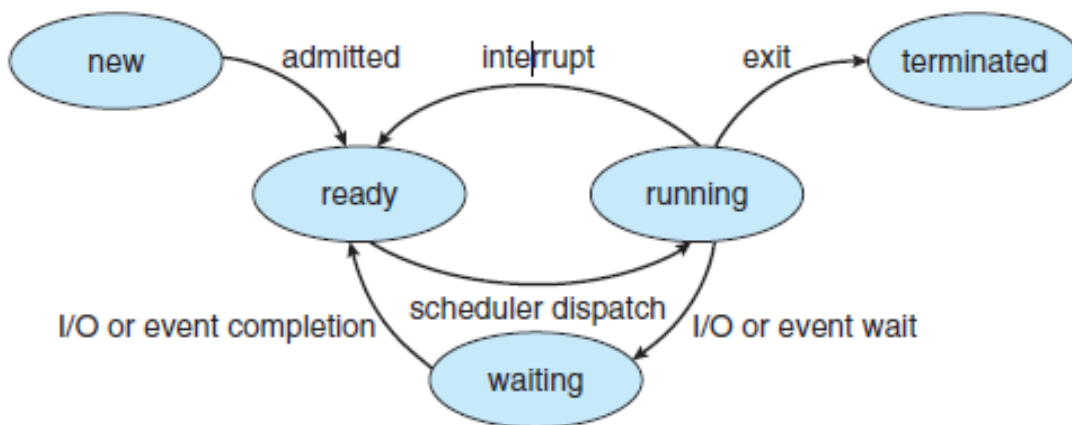
Running : 프로세서가 프로세스의 코드를 실행한다.

Waiting : 프로세스가 특정 이벤트(예. I/O 응답)가 발생할 때까지 기다린다.

Ready : 프로세스가 생성되어 Ready Queue에서 프로세서에 할당 될 때 까지 기다린다.

Terminated : 프로세스의 실행이 종료되어 할당된 프로세서를 반납한다.

프로그램을 실행하면 New Queue에 배치되며 Short-term scheduler에 의해 선택된 프로그램은 메모리에 Load된다. 메모리에 Load된 프로그램을 프로세스라고 하며 이 때 상태는 Ready가 된다. 프로세스는 Ready Queue에 배치되며 대기가 끝나고 나면 프로세서에 할당되어 Running 상태가 된다. 프로세스는 기술된 코드에 따라 실행되며 I/O입출력이나 Interrupt 신호에 의해 Ready 상태로 돌아간다. 프로세스의 실행이 종료되면 Terminated 상태가 되며 메모리와 프로세서 할당을 반납한다.



PCB

PCB(Process Control Block)은 운영체제가 프로세스를 제어하기 위해 정보를 저장해 놓은 곳이며 프로세스의 상태 정보를 저장하는 자료구조이다. PCB는 시분할 시스템을 위해 고안되었다. 프로세서에 현재 프로세스와 다음 프로세스가 교체되기 위해서는 프로세스의 정보를 저장과 불러올 필요가 있다. 이를 Context Switching이라 하며 이때 PCB를 사용한다. 운영체제에 따라 PCB의 내용은 다를 수 있으나 일반적인 내용은 다음과 같다.

Process State : 현재 프로세서의 상태를 정의한다. 상태는 new, ready, running, waiting, terminated 등이 있다.

Process number : 프로세스의 고유 번호

Program counter : 다음 실행될 프로세스의 번호

Registers : 레지스터 관련 정보

List of open files : 현재 I/O 작업 중인 파일

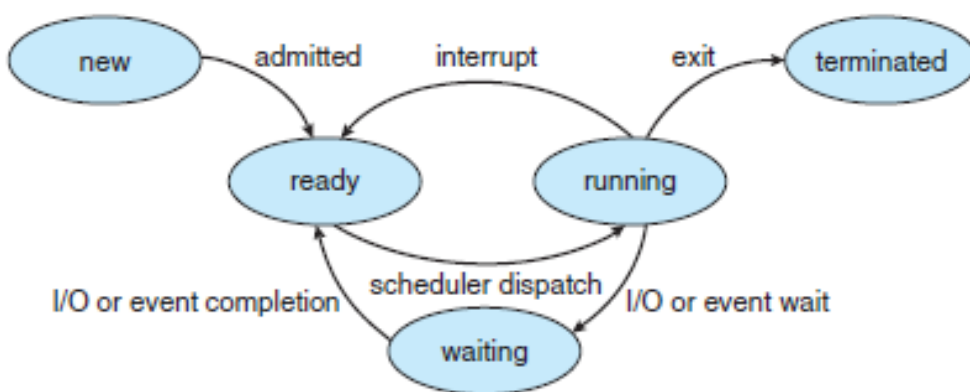
Process Scheduling 목적

다중 프로그래밍의 목적은 **CPU 활용도를 극대화**하기 위해 항상 일부 프로세서를 실행한다. 시분할의 목적은 프로세스 간에 **CPU를 자주 전환**하여 사용자가 실행 중인 각 프로그램과 상호 작용할 수 있도록 하는 것이다. 이러한 목표를 달성하면서 프로세스를 효율적으로 프로세서에 할당하는 것이 프로세스 스케줄링이다. 운영체제에서는 프로세스 스케줄링 큐에서 모든 PCB들을 관리한다. 각 프로세스의 상태에 대해 별도의 큐를 가지고 있으며 동일한 상태에 있는 프로세스들을 동일한 큐에 배치한다. 프로세스의 상태가 변경되면 PCB가 현재 대기열에서 연결 해제되고 새로운 상태 대기열로 이동한다. Queue의 종류는 다양하며 대표적으로 Job Queue, Ready Queue, Waiting Queue가 있다.

Job Queue는 실행할 작업이 배치된 자료구조이며 Long-term Scheduler에 의해 유지된다. Ready Queue는 준비 및 실행 대기 중인 모든 프로세스들을 유지한다. 새로운 프로세스는 항상 이 큐에 배치된다. Waiting Queue는 현재 CPU에 할당된 프로세스가 I/O를 사용할 수 없어 대기 중인 프로세스들을 유지한다.

Long-term Scheduler

Long-term Scheduler는 new queue에 있는 프로세스중 ready queue에 들어 갈 프로세스를 정하며 Job scheduler라고도 불린다. Long-term Scheduler의 주요 목표는 **I/O 바운드 및 CPU 바운드와 같은 균형 잡힌 프로세스 조합을 결정하는 것**이다. I/O 바운드란 연산에 수행하는 시간 보다 I/O에 더 많은 시간을 할애하는 것이다. 반면에 프로세스 바운드란 I/O 요청의 빈도를 줄이고 CPU 연산에 더 많은 시간을 할애하는 것이다. 모든 프로세스가 I/O 바운드에 할당된다면 Ready Queue에는 항상 비어 있을 것이고 Short-term Scheduler의 수행 비중은 떨어지게 된다. 반면에 모든 프로세스가 CPU 바운드에 할당된다면 I/O waiting Queue가 항상 비어 있을 것이고 장치는 사용되지 않을 것이다. 이는 시스템의 불균형을 야기한다. 또한 Long-term Scheduler는 멀티 프로그래밍 정도를 제어하여 안정적인 상태를 유지합니다. **안정적인 상태란 프로세스의 평균 생성 속도와 실행이 완료되어 시스템에서 떠나는 프로세스의 평균 이탈 속도가 같은 상태를 의미한다.** 어떤 시스템에서는 Long-term Scheduler를 사용하지 않거나 최소한으로 운용한다. UNIX나 Windows와 같은 시분할 운영체제에서는 Long-term Scheduler를 운용하지 않는다. 이로 인해 새로 들어온 모든 프로세스는 메모리에 할당된다. 다중 사용자를 지원하는 UNIX는 메모리가 물리적 한계에 다다르거나 성능이 저하될 경우 사용자에게 제한을 둘 수밖에 없다.



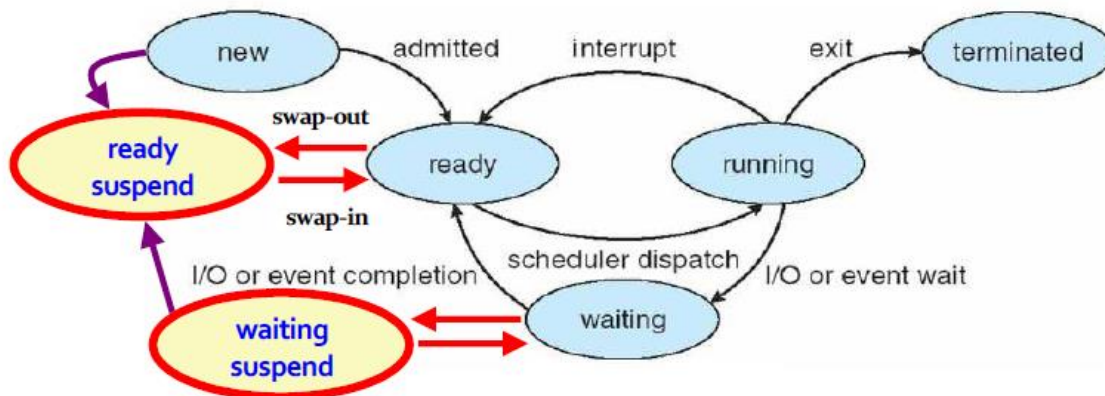
Short-term Scheduler

Short-term Scheduler는 Ready Queue에서 실행 준비가 된 프로세스 선택하여 프로세서에 할당합니다. 다음에 실행할 프로세스는 스케줄링 알고리즘으로 결정된다. 스케줄링 알고리즘은 모든 프로세스가 적절하게 실행되기 위해 중요하다. 특히 프로세스의 실행이 매우 긴 경우 다른 프로세스는 오랜 시간 Ready Queue에 대기를 하여야 한다. 이것은 Short-term Scheduler가 실행할 프로세스를 선정할 때 매우 중요함을 의미한다. 프로세스는 I/O 요

청을 대기하기 전에 몇 밀리초 동안 실행될 수 있다. 이처럼 프로세스의 실행 시간이 매우 짧기 때문에 Scheduler가 빨라야 한다. 그러므로 Short-term Scheduler는 100밀리초 마다 적어도 한번 실행된다. 100밀리초 동안 프로세스를 실행하도록 결정하는데 10밀리초가 걸린다면 총 110초를 위해 10초의 시간이 소요되었으므로 9%의 CPU낭비가 발생된다.

Medium-term Scheduler

Medium-term Scheduler란 **Swapping** 관련 스케줄링 작업을 수행한다. Swap-out은 ready 상태에서 running상태로 넘어가지 못하거나 waiting 상태에서 ready 상태로 넘어가지 못하는 상황에서 프로세스를 메모리가 아닌 임시로 하드디스크로 보내는 것을 말한다. Swap-in은 Swap-out으로 보내진 프로세스를 다시 메모리로 돌아오는 것을 말한다.



Swap-Out	Swap-In	Swap-Out
waiting → waiting suspend: 보통 선호됨.	ready suspend → ready: <ul style="list-style-type: none"> ready가 없는 경우. ready 보다 우선순위↑. 	running → ready suspend: 방금 깨어 난 waiting suspend에 의해 preemption 되는 경우.
ready → ready suspend: <ul style="list-style-type: none"> ready가 아주 큰 경우. 곧 깨어날 waiting이 ready 보다 우선순위↑. 	waiting suspend → waiting: 곧 깨어날 waiting suspend가 ready suspend 보다 우선순위↑.	

15/43

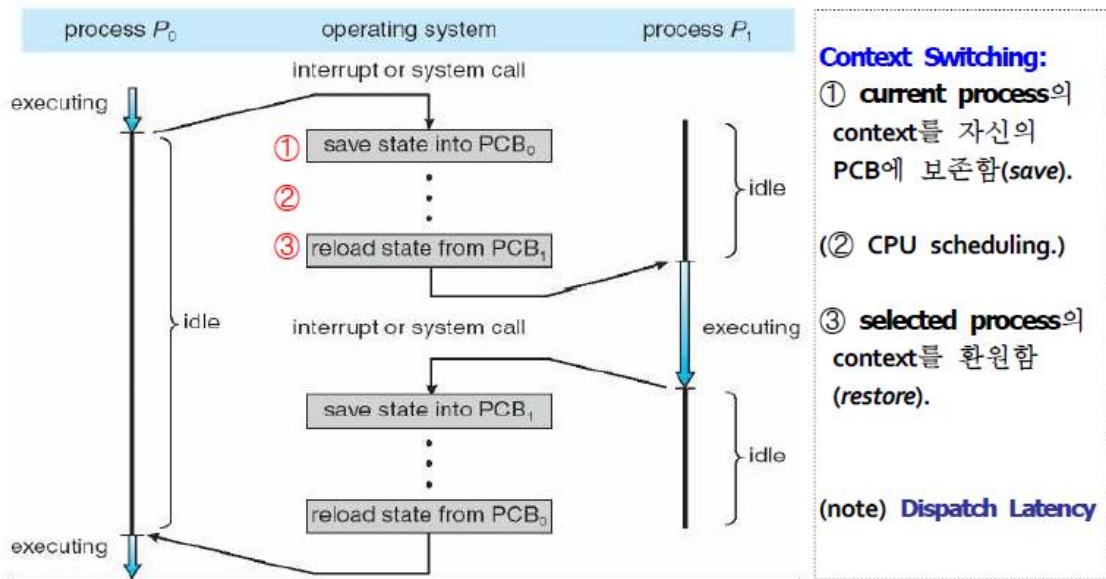
Preemption : 선점 = 미리 취하는 것

Context-switching

Context Switching이란 하나의 프로세스가 CPU를 사용 중인 상태에서 다른 프로세스가 CPU를 사용하도록 하기 위해, 현재 작업 중인 프로세스의 상태 정보를 PCB에 저장하고 새로운 프로세스의 PCB 내용을 토대로 새로운 프로세스 실행을 이어서 하는 것이다. Context란 CPU가 해당 프로세스를 실행하기 위한 해당 프로세스의 정보들이다. 이 Context는 PCB에 저장된다. 즉 Context를 불러오고 저장하기 위해서는 CPU가 작업을 하여야 하며 이는 Context Switching 시간 동안은 CPU가 어떤 작업도 할 수 없는 것이다. 이는 오버헤드라고 볼 수 있으며 CPU의 작업 효율을 떨어트린다. Context Switching이 발생하는 요인은 여러가지가 있다. I/O 요청이 들어올 때, CPU 사용 사용시 끝났을 때, 자식 프로세스를 생성할 때, 인터럽트 처리를 기다릴 때 등 그 요인은 다양하다. Context Switching 시간은 하드웨어에 의존적이다. 일부 프로세서는 여러 레지스터를 사용한다. 여기서 Context Switching의 역할은 새로운 포인터를 현재 레지스터 집합에 변경하는 것이다. 만약 레지스터에 많은 프로세스들의 데이터가 들어있어 레지스터의 데이터를 메모리로 옮기는 과정이 필요하다. 이처럼 Context Switching은 복잡한 시스템의 경우 수행해야 하는 작업의 양이 증가한다.

□ Context Switching

CPU dispatcher: Context switching + User mode로 전환 + next instruction으로 jump.



CPU Dispatcher

Dispatcher란 CPU의 제어권을 Short-term scheduler에 의해 선택된 Process에게 넘겨주는 일을 수행한다. 이는 scheduler와 CPU를 관리하는 운영체제의 핵심 기능이다. Dispatcher는 문맥교환, 사용자 모드로 전환, 사용자 프로그램을 재시작하기 위해 메모리의 적절한 위치로 이동이라는 3가지 기능을 수행한다. Dispatcher의 수행 예시는 다음과 같다.

각각 P_1 , P_2 , P_3 , P_4 프로세스가 Ready Queue에 있다. FIFO를 기준으로 Short-term scheduler에 의해 P_1 프로세스가 선택된다. 그럼 dispatcher가 P_1 을 Ready Queue에서 제거하고 CPU에 할당하여 Running 상태로 둔다. Scheduler는 P_2 를 다음 순번으로 지정할 것이다. 그러면 P_2 를 dispatcher가 CPU에 할당한다. P_3 와 P_4 도 이러한 방식으로 진행된다.

프로세스 생성과 종료

Fork

Fork란 자신의 프로세스를 복제하여 자식 프로세스를 생성하는 것을 의미한다. 프로세스가 다른 프로그램을 실행하기 위해서 먼저 자신의 프로세스를 복제한다. 그러면 자식 프로세스라고 불리는 복사본은 exec 시스템 호출을 통하여 다른 프로그램과 함께 자신의 프로세스를 겹쳐 놓는다. Fork는 자식 프로세스를 위한 별도의 주소 공간을 할당한다.

Fork()를 수행하면 반환 값이 주어지는데 부모프로세스는 **fork() > 0** 이며 자식프로세스는 **fork() = 0**으로 반환 된다.

프로세스가 Fork를 호출할 때, 그것은 부모 프로세스로 간주되고 새로 생성된 프로세스는 자식이다. Fork가 끝난 후, 두 프로세스 모두 같은 프로그램을 실행할 뿐만 아니라, 마치 두 프로세스 모두 시스템 콜을 불렀던 것처럼 실행을 재개한다. 그런 다음 시스템 콜의 반환 값을 검사하여 자식 혹은 부모 프로세스의 상태를 확인하고 그에 따라 조치를 취한다.

부모와 자식프로세스와의 관계는 자원공유, 실행, 주소공간 관점에서 볼 수 있다.

자원 공유 - 부모의 자원을 전부 공유 vs 일부 공유 vs 공유 안함

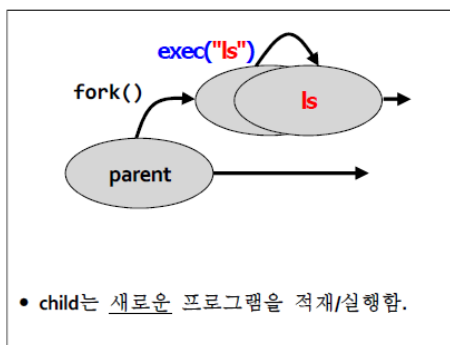
실행 - 부모와 자식 동시 실행, 자식이 종료될 때 까지 대기

주소 공간 - 부모와 동일한 프로그램 실행 vs 별도 프로그램 실행

Exec

Exec이란 이전의 실행 파일을 대체하면서, 이미 존재하는 프로세스의 문맥(메모리에 할당된 데이터)에서 executable file을 실행하는 운영 체제의 기능을 의미한다. 즉 Exec를 호출한 프로세스의 PID가 그대로 새로운 프로세스에 적용이 되며, Exec를 호출한 프로세스는 새로운 프로세스에 의해 덮어 쓰여진다. 이는 새로운 프로세스를 메모리에 할당하는 Fork와는 차이점을 보인다.

- **exec(new_program)** : 새로운 프로그램을 적재하고 실행함.



```
int main() {
    pid_t;
    ...
    pid = fork();
    if (pid == 0) {
        execlp("/bin/ls", ls, NULL);
    } else {
        부모 프로세스가 실행.
    }
    부모 프로세스가 실행.
}
```

pid == 0 이면, 즉 자식 프로세스이면 exec를 실행하고 pid가 그 이외의 값이라면 부모 프로세스가 실행된다.

Wait

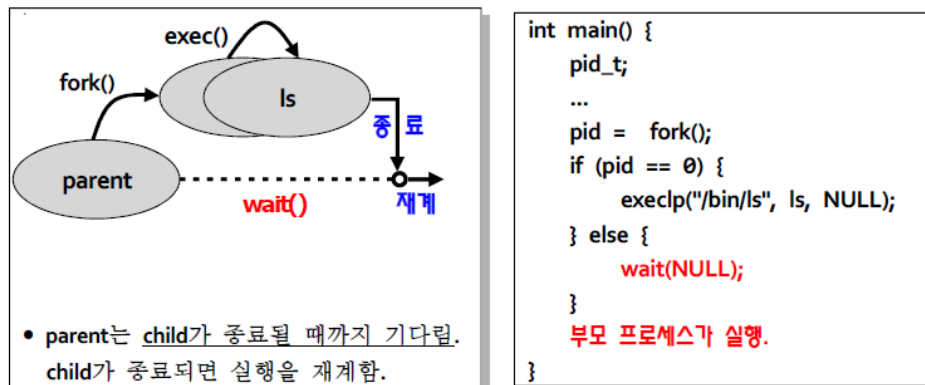
컴퓨터 운영 체제에서는 프로세스(또는 태스크)가 실행을 완료하기 위해 다른 프로세스를 기다릴 수 있다. 대부분의 시스템에서 부모 프로세스는 독립적으로 실행되는 자식 프로세스를 만들 수 있다. 그런 다음 **부모 프로세스는 자식 프로세스가 실행되는 동안 부모 프로세스의 실행을 중지하는 대기 시스템 호출을 실행할 수 있다.** 자식 프로세스가 종료되면 운영 체제에 종료 상태를 반환한 다음 대기 중인 상위 프로세스로 돌아간다. 그런 다음 상위 프로세스가 실행을 재개한다.

또한 현대의 운영체제는 프로세스의 스레드가 다른 스레드를 생성하여 유사한 방식으로 종료될 때까지 기다리는 시스템 호출을 제공한다.

운영 체제는 프로세스가 자식 프로세스가 종료될 때까지 기다리거나 특정 단일 자식 프로세스(프로세스 ID로 식별된)가 종료될 때까지 기다릴 수 있는 다양한 대기 호출을 제공할 수 있다.

자식 프로세스에 의해 반환되는 종료 상태는 일반적으로 그 프로세스가 정상적으로 종료되었는지 비정상적으로 종료되었는지 여부를 나타낸다. 정상적인 종료의 경우, 이 상태에는 프로세스가 시스템으로 반환한 종료 코드(일반적으로 정수 값)도 포함된다.

- **wait()** : parent는 child가 종료될 때까지 대기함.



고아(Orphan) 프로세스

고아 프로세스란 부모 프로세스가 자식 프로세스보다 먼저 종료되면 자식 프로세스는 고아가 되는 경우를 말한다. 부모 프로세스가 자식 프로세스보다 먼저 종료되면 init 프로세스가 자식 프로세스 새로운 부모 프로세스가 된다. 종료되는 프로세스가 발생할 때 커널은 이 프로세스가 누구의 부모 프로세스인지 확인한 후, 커널이 자식 프로세스의 부모 PID를 1 (init 프로세스)로 바꿔준다. Init 프로세스란 유닉스 계열 운영체제에서 부팅 시 최초로 생성되는 프로세스이며 종료될 때까지 계속 살아있는 데몬 프로세스이다.

고아 프로세스가 작업을 종료하면 init 프로세스가 wait함수를 호출하여 고아 프로세스의 종료 상태를 회수함으로써 좀비 프로세스가 되는 것을 방지한다.

좀비(Zombie) 프로세스

좀비 프로세스란 자식 프로세스가 종료되었지만 부모 프로세스가 자식 프로세스의 종료 상태를 회수하지 않아 자식 프로세스가 메모리에서 사라지지 않는 상태를 말한다.

프로세스 종료

프로세스가 종료되는 경우는 다음과 같다.


1. 프로세스가 `exit()`를 호출
2. 부모 프로세스가 자식 프로세스를 종료 = Abort
3. Descendents를 가지는 부모 프로세스가 종료되는 경우 = 고아 프로세스의 처리 방법

프로세스 간 통신

IPC : Inter-Process Communication

기본적으로 프로세스들은 서로 독립적으로 실행된다. 하지만 한 프로세스가 다른 프로세스로 부터 영향을 받거나 영향을 끼친다면, 자원 공유, 모듈화, 병렬 처리, 편리성을 위해 프로세스끼리 통신을 할 필요가 있다. 이러한 상황에서 사용하는 것이 IPC(프로세스 간 통신)이다.

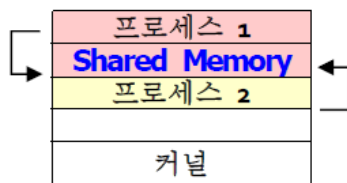
IPC Model

1. 공유 메모리 모델 (Shared Memory Model)
2. 메시지 전송 모델 (Message Passing Model)
3. 소켓 (Socket )
4. 원격 프로시저 호출 (RPC : Remote Procedure Call)
5. 파이프 (Pipe)
6. 기타 등등...

1. Shared Memory Model

공유 메모리란 프로세스간 메모리 영역을 공유해서 사용할 수 있도록 허용하는 것을 말한다. 프로세스가 공유메모리 할당을 커널에 요청하면 커널은 해당 프로세스에 메모리 공간을 할당해준다. 이후 어떤 프로세스 건 해당 메모리 영역에 접근할 수 있다.

□ Shared Memory Model

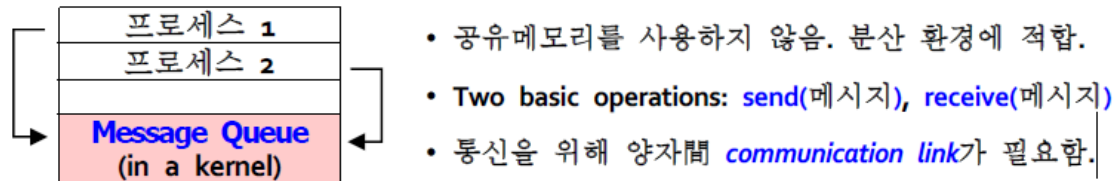


- ① 한 프로세스가 자신의 주소공간에 **Shared Memory** 설정함. (시스템호출)
- ② 다른 프로세스는 공유메모리를 자신의 주소공간에 **attach**. (시스템호출)
- ③ 읽고 씀 (note) 동기화 Synchronization

2. Message Passing Model

메시지 전송 모델은 프로세스간 연결 없이 메시지 queue를 통해서 데이터를 읽고 쓰는 것을 말한다. 밑의 그림처럼 프로세스 1과 2는 메시지 queue를 이용하여 데이터를 저장하고 읽을 수 있다.

□ Message Passing Model



메시지 전송 모델에는 **직접 통신(Direct Communication)**과 **간접 통신(Indirect Communication)**이 있다. 직접 통신은 프로세스 1이 커널에 메시지를 직접 전송하면 커널이 프로세스 2에게 메시지를 직접 전송하는 방식이다. 직접 통신을 위해서는 프로세스 간 일대일 통신이 되어야 한다. 간접 통신은 프로세스 1이 커널에게 특정 메시지 queue에 메시지를 저장한다고 말하면 프로세스 2가 그 메시지 queue로 가서 읽어오는 방식이다. 즉 메시지 전송 모델은 직접 전송하는지 안하는지로 나뉜다.

• 송신자, 수신자의 명명(naming)

직접 통신	간접 통신
send(수신자, msg) receive(송신자, msg)	send(메일박스, msg) receive(메일박스, msg)
<ul style="list-style-type: none"> - sender, receiver 명시. - 일대일 통신. (전용 link) 	<ul style="list-style-type: none"> - mailbox(or port)를 공유하는 프로세스間 통신. - 다대다 통신
Asymmetry 방식: send(수신자, msg) receive(msg)	system이 수신할 receiver 선택 방법: <ul style="list-style-type: none"> - 최대 두 프로세스 間 link 설정. - 한순간 오직 하나의 프로세스만 receiver() 실행 허용. - 임의의 receiver 선정. (note) 선정 알고리즘에 따름.

메시지 전송 모델의 장점

1. 공유 메모리 모델보다 구현하기가 쉬움
2. 통신 지연 시간이 길기 때문에 메시지 전달 모델을 사용하여 병렬 하드웨어를 구축하기 쉽다.

메시지 전송 모델의 단점

연결 설정에 시간이 걸리기 때문에 메시지 전달 모델이 공유 메모리 모델보다 통신속도가 느리다.

• 송수신 동기화 (synchronization)

송신	동기적	Sender is blocked until msg is received by receiver or mailbox.
	비동기적	Sender sends msg and resumes operation.
수신	동기적	Receiver blocks until a msg is available.
	비동기적	Receiver retrieves either a valid msg or a null.

- 임의의 조합으로 통신 가능함.
- 랑데뷰 **Rendezvous** = blocking send + blocking receiver.

Rendezvous

프로세스간 통신에서 send와 receive가 blocking방식인 경우 sender와 receiver간을 Rendezvous라고 한다. Blocking 방식이란 Sender가 메시지를 전송하면 receiver로부터 응답이 도착할 때까지 대기하여야 한다. 반대로 Receiver는 메시지 올 때까지 대기한다.

Blocking방식에서 다른 클라이언트 간 영향 없이 진행되기 위해서는 서버와 클라이언트들 간에 별도의 쓰레드를 생성하여야 한다. 클라이언트의 수가 늘어날수록 쓰레드의 수는 많아지며 CPU의 문맥교환 횟수가 증가하게 된다. 이 때문에 CPU의 효율을 떨어트리게 된다.

다른 통신 방법으로는 비동기 방식이 있다. 이는 sender가 메시지를 송신하고 다른 작업을 진행하고 receiver가 메시지를 수신하거나 없더라도 작업을 계속 수행한다.

Buffering

메시지를 송수신할 때는 임시 Queue를 사용하는데 이를 buffering이라고 한다. Buffering의 구현 방식에는 3가지가 있다.

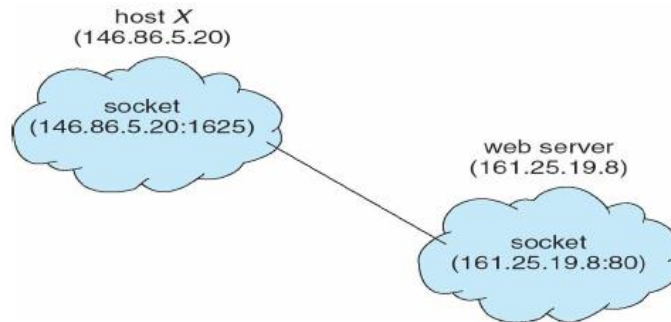
1. Zero capacity : Queue의 최대 길이가 0이다. 그러므로 어떤 메시지도 Queue에 대기할 수 없다. 이 경우 sender는 receiver가 메시지를 수신할 때까지 대기한다.
2. Bounded capacity : Queue의 길이는 N이다. 최대 N개의 메시지가 Queue에 있을 수 있다. Sender가 새 메시지를 보낼 때 Queue의 대기열이 가득 차 있지 않으면 그 메시지는 Queue에 배치되고 대기할 필요 없이 계속해서 작업을 진행할 수 있다.
3. Unbounded capacity : Queue의 길이는 무한하다. 따라서 어떤 수의 메시지라도 Queue에 대기할 수 있다.

• 버퍼링 (Buffering) - 통신(직접, 간접) 시 사용되는 임시 큐의 유형:

	큐의 길이	송신 프로세스의 blocking
Zero capacity	zero.	수신자가 수신할 때까지 (랑데뷰)
Bounded capa.	유한 길이.	큐가 full일 때
Unbounded capa.	무한 길이.	Never blocks.

3. Socket

Socket이란 컴퓨터 네트워크를 경유하는 프로세스 간 통신의 종착점이다. Socket은 포트 번호와 연결된 IP주소로 식별된다. 일반적인 Socket은 클라이언트-서버 구조를 사용한다.



서버는 특정 포트로부터 읽은 클라이언트 요청이 올 때까지 기다린다. 한번 요청을 받으면 서버는 클라이언트와의 연결을 완료하기 위해 연결을 받아들인다. 특정 서비스(telnet, FTP, HTTP 등)를 구현하는 서버는 well-known port를 수신한다. 1024이하의 모든 포트는 well-known port로 알려져 있다.

TMI : Java에서는 3가지 소켓 방식을 제공한다. Connection-oriented (TCP) socket은 Socket 클래스로 구현된다. Connectionless (UDP) socket은 DatagramSocket 클래스를 이용한다. 마지막으로 MulticastSocket 클래스는 DatagramSocket 클래스의 하위 클래스이다.

4. RPC : Remote Procedure Calls

RPC는 분산 네트워크 환경에서 Socket의 단점을 보완하여 프로그래밍을 쉽게 하도록 한다. 클라이언트-서버 간의 통신에 필요한 상세 정보는 최대한 감추고, 클라이언트는 일반 메소드를 호출하는 것처럼 요청하고, 서버는 도 마찬가지로 일반 메소드 다루듯 응답을 해준다.

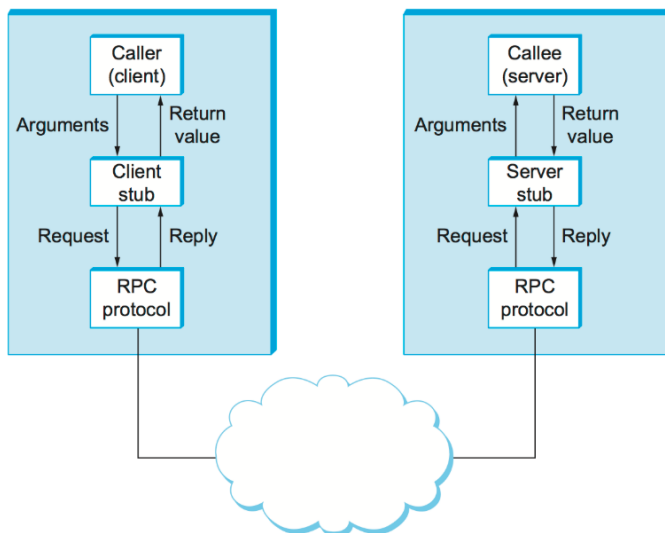
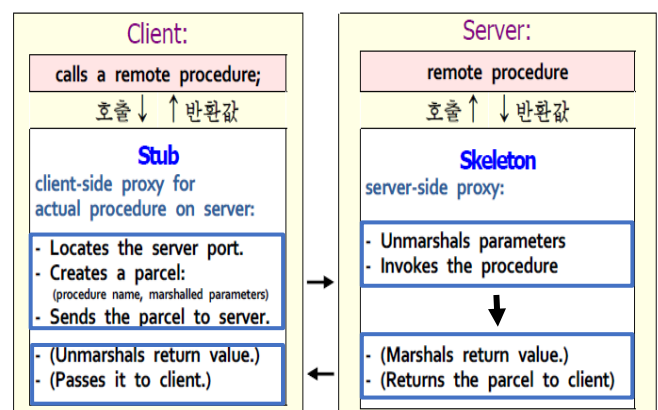


Figure 2. Complete RPC mechanism.

Remote Procedure Call (RPC)

- Abstracts Procedure Calls between processes on networked systems.



marshal = gather people or things together and arrange them for a particular purpose
unmarshal = decode from a marshalled state.

Stub이란 그 자체로는 기능을 발휘하지 못하지만 원래 개체가 무엇인지 알아낼 수 있는 참조 역할을 하고, 원 개체의 기능 조작을 위임받는 위임자(Proxy) 역할을 하는 작은 개체이다.

서비스를 제공하는 쪽을 Stub, 서비스를 제공받는(요청하는) 쪽을 Skeleton이라 한다.

5. Pipe

Pipe는 두 프로세스가 서로 소통할 수 있게 하는 통로 역할을 한다. Pipe의 구현에는 꼭 고려할 4가지 측면이 있다.

1. Pipe를 단방향(unidirectional) 혹은 양방향(bidirectional)으로 할 것인가?
단방향은 서로에 대해 한쪽은 읽기만 한쪽은 쓰기만 가능한 통신이다.
2. 양방향 통신을 지원한다면 전이중(half duplex) 혹은 반이중(full duplex) 어떤 것을 할 것인가?
전이중은 동시 송수신이 가능하며 반이중은 동시 송수신이 불가능 하다.
3. 프로세스간 통신에 관계(예. 부모-자식)가 있는가?
4. Pipe가 네트워크를 통해 통신할 수 있는가? 혹은 로컬에서만 통신하는가?

Pipe의 종류는 Ordinary Pipe, Name Pipe 2가지이다.

- **Ordinary Pipe**

- only unidirectional
- 오직 parent-child 간 통신만 가능 (note) parent가 생성하고 child는 상속함.
- 파일처럼 취급함: (Unix) file read, write 시스템호출을 사용하여 통신함.

- **Named Pipe**

- bidirectional
- 임의의 프로세스들 간 통신 가능

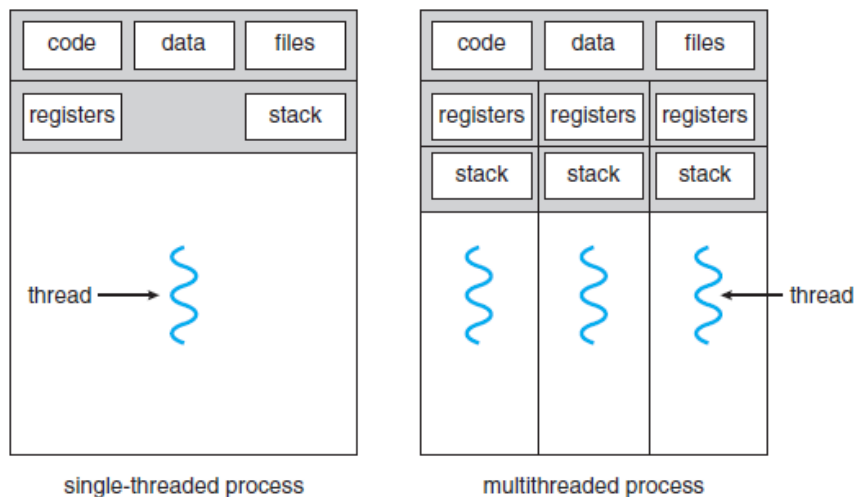
Unix	Windows
<ul style="list-style-type: none">• byte 단위 전송• half duplex 통신만 가능• local 내 통신만 가능	<ul style="list-style-type: none">• byte 및 message 단위 전송• full duplex 통신도 가능• remote 간 통신도 가능
<ul style="list-style-type: none">• FIFO라 부름• 파일시스템에 정규 파일처럼 존재함<ul style="list-style-type: none">- persistent- 생성: mkfifo() 시스템호출- 사용: open(), read(), write(), close() 시스템호출	

Thread

3장에서 다룬 프로세스는 단일 제어 스레드를 가지는 실행 프로그램이다. 모든 현대 운영체제에서는 다중 제어 스레드를 포함하는 프로세스를 가능하게 하는 기능을 제공한다. 스레드란 CPU 활용의 기본적인 단위(unit)이다. 스레드는 같은 프로세스에 속해 있는 프로세스의 코드 영역, 데이터 영역 그리고 열린 파일과 신호와 같은 운영 체제의 자원을 다른 스레드와 공유한다. Heavyweight 프로세스는 단일 제어 스레드를 가지고 있다.

그럼 다중 프로세스와 다중 스레드의 차이점은 무엇일까?

멀티 프로세스와 멀티 스레드는 양쪽 모두 여러 작업이 동시에 진행된다는 공통점을 가지고 있다. 하지만 멀티 프로세스에서 각 프로세스는 독립적으로 실행되며 각각 별개의 메모리를 차지하고 있는 것과 달리 멀티 스레드는 프로세스 내의 메모리를 공유해 사용할 수 있다.



1 스레드

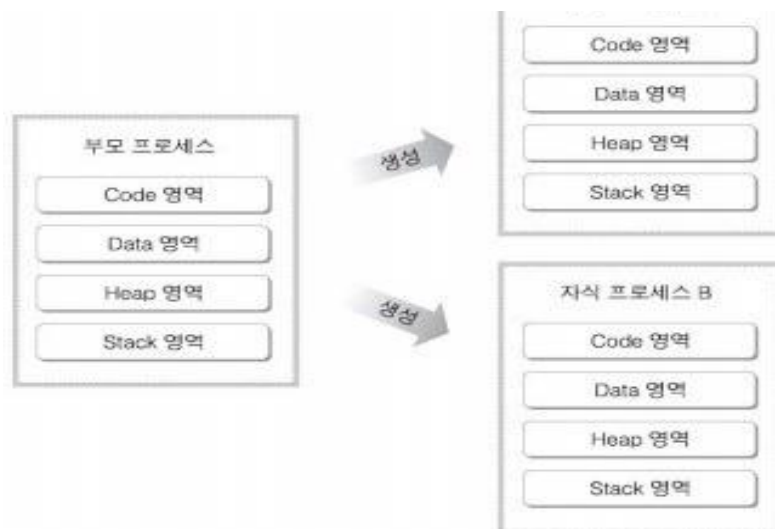
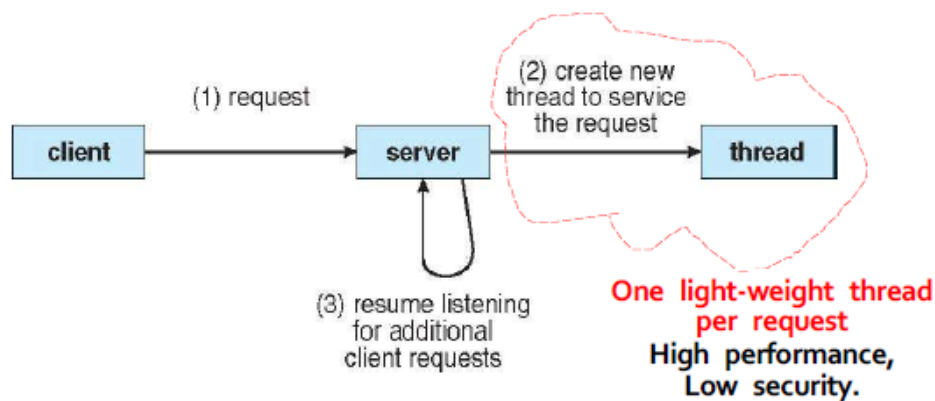


그림 1 멀티 프로세스

멀티 스레드 장점

1. **반응성** - 상호 작용하는 응용 프로그램을 멀티 스레딩하면 프로그램의 일부가 차단되거나 긴 작업을 수행하더라도 프로그램이 계속 실행되도록 하여 사용자에게 대한 응답성을 높일 수 있다.
2. **자원 공유** - 프로세스들은 오직 자원을 공유 메모리나 메시지 교환 같은 기술을 통해 공유할 수 있다. 이러한 기술은 프로그래머에 의해 명확하게 기술되어야 하지만 스레드는 기본적으로 그들이 속한 프로세스의 자원과 메모리를 공유할 수 있다.
3. **경제성** - 프로세스 생성에 메모리와 자원을 할당하는 것은 비용이 높다. 스레드들은 그들이 속한 프로세스의 자원을 공유하기 때문에, 스레드를 생성하고 문맥 교환하는 것이 더 경제적이다.

• Multi-threaded server architecture



(cf) Single-threaded process로 구현하는 경우

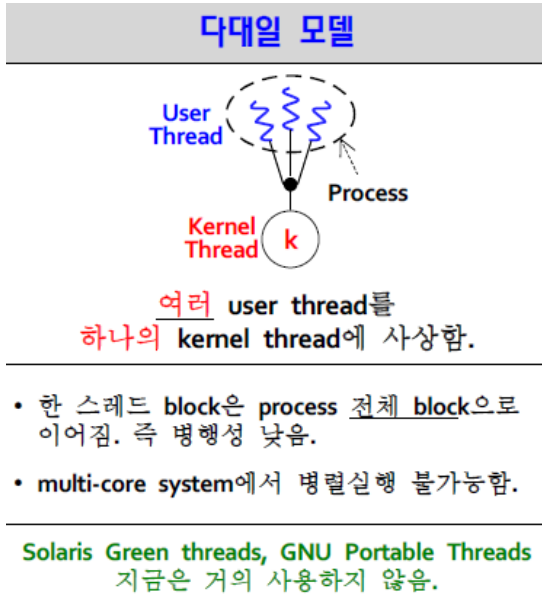
```
do forever {  
    get a request;  
    if( pid=fork() == 0 ) { // one heavy-weight process per request  
        exec("요청처리프로그램");  
    }  
}
```

4. **확장성** - 멀티 프로세서 아키텍처에서, 멀티 스레딩의 이점은 다른 프로세서에서 스레드가 동시에(병렬로) 실행될 수 있다는 점이 훨씬 크다. 싱글 스레드 프로세스는 오직 하나의 프로세서에서만 실행될 수 있다.

멀티 스레드 모델

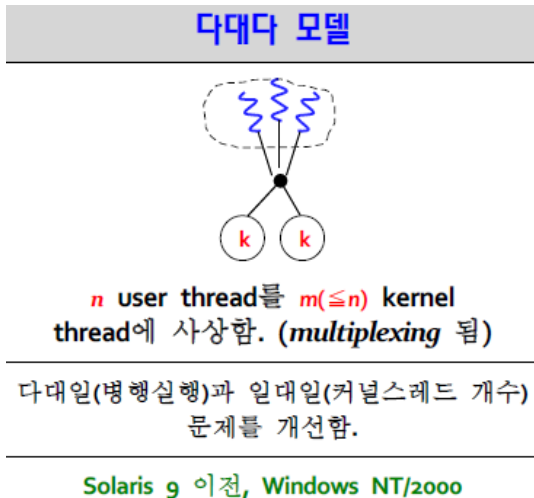
멀티 스레드는 Kernel thread와 User thread를 지원한다. Kernel 스레드는 운영체제가 작업 관리하는 스레드이며, Kernel-supported thread 또는 **Lightweight process**라고 불린다. User 스레드는 사용자가 생성 · 관리하는 스레드이다. 운영체제는 kernel 스레드 만 scheduling을 하므로 user 스레드가 실행되기 위해서는 **kernel thread에 사상(Mapping)되어야 한다.**

1. 다대일 모델(Many-to-One Model)



여러 개의 사용자 수준 스레드들이 하나의 커널 스레드(프로세스)로 매핑되는 방식으로, 사용자 수준에서 스레드 관리가 이루어진다. 주로 커널 스레드를 지원하지 않는 시스템에서 사용하며, **한 번에 하나의 스레드만이 커널에 접근할 수 있다는 단점**이 있다. 하나의 스레드가 커널에 시스템 호출을 하면 나머지 스레드들은 대기해야 하기 때문에 진정한 의미의 동시성을 지원하지 못한다. 다시 말해, 여러 개의 스레드가 동시에 시스템 호출을 사용할 수 없다. 또한 커널 입장에서는 프로세스 내부의 스레드들을 인식할 수 없고 하나의 프로세스로만 보이기 때문에 다중처리기 환경이라도 여러 개의 프로세서에 분산하여 수행할 수 없다.

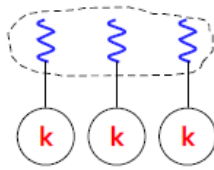
2. 다대다 모델(Many-to-Many Model)



여러 개의 사용자 스레드를 여러 개의 커널 스레드로 매핑시키는 모델이다. 다-대-일 방식과 일-대-일 방식의 문제점을 해결하기 위해 고안되었다. 커널 스레드는 생성된 사용자 스레드와 같은 수 또는 그 이하로 생성되어 스케줄링한다. 다-대-일 방식에서 스레드가 시스템 호출시 다른 스레드가 중단되는 현상과 일-대-일 방식에서 사용할 스레드의 수에 대해 고민하지 않아도 된다. 커널이 사용자 스레드와 커널 스레드의 매핑을 적절하게 조절한다.

3. 일대일 모델(One-to-One Model)

일대일 모델



하나의 user thread를
하나의 kernel thread에 사상함.

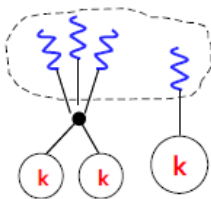
- 다대일 보다 높은 병행성.
- 병렬실행 가능함.
- 커널 스레드 개수 증가 (성능 저하 야기)
- 시스템 내 thread 개수 제한함.

Solaris 9 부터, Linux, Windows

사용자 스레드들을 각각 하나의 커널 스레드로 매핑시키는 방식이다. 사용자 스레드가 생성되면 그에 따른 커널 스레드가 생성되는 것이다. 이렇게 하면 다-대-일 방식에서 시스템 호출 시 다른 스레드들이 중단되는 문제를 해결할 수 있으며 여러 개의 스레드를 다중처리기에 분산하여 동시에 수행할 수 있는 장점이 있다. 그러나 커널 스레드도 한정된 자원을 사용하므로 무한정 생성할 수는 없기 때문에, 스레드를 생성할 때 그 개수를 염두에 두어야 한다.

4. Two-Level Model

Two-level Model



다대다에 일대일 결합한 방식

다대다 모델의 변형

Solaris 9 이전, Tru64UNIX

스레딩 방식

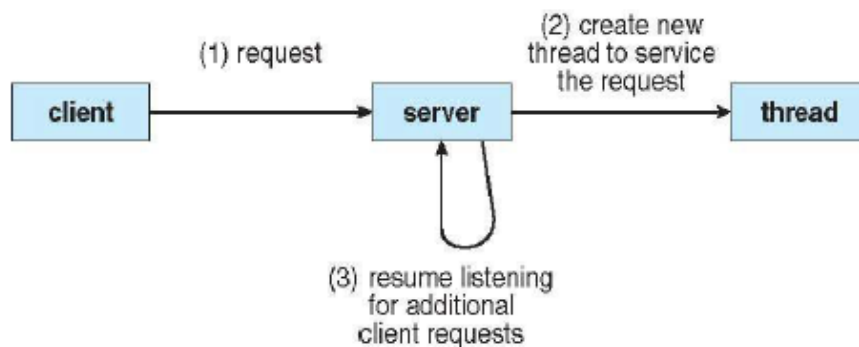
비동기 스레딩	<ul style="list-style-type: none"> - 부모는 자식 생성 후 즉시 실행 재개함. - 모든 스레드는 독립적으로 실행되며 (데이터 공유 거의 없음), 부모는 자식의 종료를 알 필요 없음. - multithreading server에서 사용되는 전략.
동기 스레딩	<ul style="list-style-type: none"> - 부모는 모든 자식 종료 후 실행을 재개함. (fork-join 전략) (자식 스레드들은 서로 독립적으로 실행됨.) - 보통 데이터 공유 많음.

Implicit Threading

멀티 프로세싱의 성장과 함께, 수백, 수천개의 스레드를 사용하는 응용 프로그램들이 늘어났다. 이러한 응용프로그램을 설계하는 것은 쉽지 않다. 다중 스레드 응용 프로그램은 스레드 생성과 관리를 개발자 수준에서 컴파일러와 런타임 라이브러리에 맡기는 것이다. 이러한 방식을 Implicit Threading이라고 하며 오늘날 대중적인 방법이다. 즉 Implicit Threading이란 스레드의 scheduling을 사용자가 하는 것이 아니라 운영체제가 또는 컴파일러가 대신하는 것이다.

- 수백-수천 개의 thread를 가지는 multi-threaded application 등장
필요할 때마다 thread 생성하면

- application의 성능 저하 (많은 스레드 생성 비용)
- system resource 고갈 염려 (unstable)



- **Implicit threading**

thread의 생성과 관리를 application에서 분리하여
compiler와 thread library가 수행하게 함:

Thread pool, OpenMP, Grand Central Dispatch, Thread Building Block,
etc.

Thread pool이란 프로세스를 시작하면서 일정 수의 스레드를 미리 pool에 생성한다. Pool에는 스레드들이 작업을 때까지 기다린다. 만약 서버가 요청을 받는다면, 서버는 pool에서 스레드를 가져와 들어온 요청을 수행한다. 스레드의 작업이 완료되면 다시 pool로 돌아가 작업이 올 때까지 대기한다. 만약 pool에 스레드가 없다면 서버는 작업을 마친 스레드가 생길 때까지 기다린다.

Thread Pool의 장점

1. 이미 있는 스레드를 가지고 요청을 수행하는 것이 새로운 스레드를 만드는 것 보다 빠르다.
즉, 응용 프로그램의 성능이 향상된다.
2. Thread Pool은 어느 한 시점에 존재하는 스레드의 수를 제한함으로써 동시에 많은 수의 스레드를 지원하기 힘든 시스템에게는 중요하다.
즉, 시스템의 안정성이 높아진다.
3. 수행할 작업을 분리하면 작업 마다 서로 다른 전략(방식)으로 처리할 수 있다. 예를 들어 어느 한 작업을 일정 시간 이후에 실행하거나 정기적으로 실행, 둘 다 설정할 수 있다.
즉, Pool의 사용 패턴을 동적으로 조정할 수 있다.

Thread 관련 문제

fork()와 exec() 시스템호출의 의미

다중 스레드 프로그램에서 fork()와 exec()은 3장의 프로세스와는 다른 개념을 보입니다. 만약 한 스레드가 fork()를 호출 한다면, 새로운 프로세스가 모든 스레드를 가지고 복제되는가 아니면 하나의 스레드만 가지고 복제되는가? 이러한 문제 때문에 fork()는 두가지 의미를 가진다.

- fork()의 두 가지 의미

- ① process 내의 모든 thread 복제

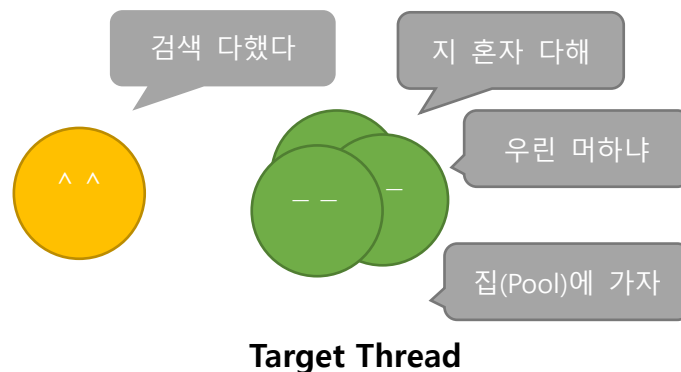
- ② fork()를 호출한 thread 만 복제 (note) fork(); exec(); 경우에 유용.

- ※ 일부 UNIX 시스템은 두 가지 버전의 fork()를 제공함.

Exec()은 3장의 개념과 똑같이 기존 프로세스에서 모든 스레드를 가지고 새로운 프로세스로 대체된다.

Thread Cancellation(스레드 실행 취소)

스레드 실행 취소란 한 스레드가 다른 스레드를 강제 종료시키는 것을 의미한다. 예를 들어 데이터베이스 검색을 한다고 하면 여러 스레드가 동시에 검색을 수행한다. 그중 한 스레드가 검색 작업을 완료하여 결과를 반환하면 현재 검색중인 다른 스레드들(target thread)은 취소될 수 있다. 또 다른 예로 웹 페이지가 로딩 중일 때 스레드는 작업을 분배하여 이미지를 로딩한다. 이때 사용자가 웹 브라우저의 취소 버튼을 누르면 모든 스레드(target thread)는 취소된다. Target thread란 취소될 스레드를 의미한다.



Thread Cancellation에는 Asynchronous cancellation(비동기 취소)와 Deferred cancellation(지연 취소)가 있다.

1. Asynchronous cancellation(비동기 취소) - 한 스레드가 target thread를 즉시 종료시킨다.
2. Deferred cancellation(지연 취소) - Target thread는 주기적으로 종료 여부를 확인하여 질서정연하게 종료할 수 있는 기회를 제공한다. 즉, 안전하게 스레드를 종료할 수 있다.