OS

1장 운영체제 소개(최종수정일 8/1)

**운영체제의 역할**

응용소프트웨어가 직접 하드웨어 제어 -> 소프트웨어 엔지니어들이 하드웨어와 하드웨어 명령어를 구체적으로 알아야 하는 문제 + 여러 응용 프로그램 수행 시 일괄적인 하드웨어 제어가 불가능해 비효율적인 리소스 관리 문제 -> 운영체제 등장 -> 하드웨어 제어, 리소스 관리와 응용 서비스의 역할 구분

**커널 모드와 사용자 모드**

응용 프로그램이 하드웨어를 직접 조작하면 시스템의 안정성이 저해되므로 커널 모드와 사용자 모드로 구분 -> 커널(슈퍼바이저) 모드에서는 CPU 명령어를 통해 하드웨어를 직접 제어할 수 있음, 사용자(보호) 모드에서는 불가능 -> 응용 프로그램은 사용자 모드에서만 동작한다.

**일체형(모놀리식, Monolithic kernel) 커널:** 운영체제의 모든 서비스 커널, 커널 내부 요소로 존재하는 여러 서비스가 효율적으로 상호작용하지만 하나의 요소라도 오류가 발생하면 시스템 전체의 장애를 불러올 수 있다.

**마이크로 커널:** 운영체제의 요소 대부분을 커널 외부로 분리, 커널 내부에서는 메모리 관리, 프로세스 간 통신(IPC;Inter Process Communication), 멀티태스킹 등 최소의 요소만 남겨놓음. 파일 시스템, 장치 드라이버, 네트워크 프로토콜 등은 사용자 모드에서 동작하도록 분리된다. 새 서비스를 추가해 운영체제를 확장하기 쉽고 커널 외부에 문제가 발생해도 커널 내부에는 영향이 없기 때문에 안정적으로 유지보수를 할 수 있다. 그러나 커널 외부에 있는 운영체제 요소들이 서로 데이터를 주고 받을 때 프로세스 간 통신을 해야만 하기 때문에 성능 저하가 발생한다.

**응용프로그램이 하드웨어 제어를 원할 경우?** 응용 프로그램이 시스템 호출을 통해 운영체제에 서비스를 요청한다. 이때 운영체제의 커널이 동작해 하드웨어를 제어하게 된다.

**운영체제의 구성:** 프로세스 관리자, 메모리 관리자, 장치 관리자, 파일 관리자

**운영체제의 변천:** 일괄처리(Batch) -> 시분할(time-sharing)-> 실시간(real-time) -> 분산(distributed)

2장 프로세스와 쓰레드

**프로세스의 구성:** 프로세스 번호(PID), 프로세스 상태, 프로그램 카운터, 레지스터, 메모리 관리 정보, 프로세스 우선순위, 회계정보

**프로세스 상태 관리:** 프로세스는 생성, 준비, 실행, 대기, 종료의 다섯 상태 중 하나의 상태에 있게 된다.

**생성:** 처음 작업이 시스템에 주어진 상태. 프로세스가 시작되면 해당 작업에 대한 프로세스 제어 블록을 생성해 작업 큐에 넣는다. 이때 PID가 결정된다. 프로세스 메모리 구조가 생성되면 준비 상태로 전이된다.

**준비:** 준비 큐에 머물면서 CPU 할당을 기다리는 상태. 스케줄러에 의해 선택되면 CPU를 할당 받아 실행상태로 전이된다. 프로세스에 CPU 점유율을 할당하는 이 과정이 디스패치(dispatch)이다.

**실행:** 프로세스가 처리되는 상태. CPU가 프로세스의 명령들을 처리하는 단계이다. 프로세스 처리 중 스케줄러가 준비 큐에서 다른 프로세스를 선택하면 실행 상태의 프로세스는 CPU를 회수당하며 준비상태로 전이된다. 스케줄링 알고리즘에 따라 CPU 할당시간이 만료된 경우 혹은 더 높은 우선순위의 프로세스가 준비 큐에 들어오는 경우 발생. 또 프로세스가 I/O나 페이지 교환을 요구하는 경우에도 대기상태로 전이한다. 입출력과 페이지 교환은 상대적으로 시간이 오래 걸리기 때문에 그동안 CPU를 다른 프로세스에 할당해 활용도를 높인다.

**대기: [**프로세스의 I/O 작업이 끝남, 특정 자원을 할당 받음]까지 실행이 보류되는 상태. 재개조건이 만족(이벤트 발생)되면 준비상태로 전이된다.

**종료:** 실행 상태의 프로세스가 모든 처리를 완료한 상태. 또는 운영체제가 에러를 감지해 프로세스를 강제로 종료시킨 상태. 종료 상태 시에 프로세스에 할당되었던 모든 자원은 운영체제가 회수한다.

**부모 프로세스와 자식 프로세스:** 사용자가 직접 실행하는 프로세스 외에 프로세스가 다른 프로세스를 실행하는 경우 시스템 호출 프로세스를 부모 프로세스, 시스템 호출을 통해 새로 생성된 프로세스를 자식 프로세스라고 한다. 프로그램 생성 시에 시스템 호출 방법은 운영체제마다 상이.

UNIX와 Linux는 fork() 시스템 호출을 통해 자식 프로세스 생성. 이 경우 자식 프로세스는 부모 프로세스의 복제본이 된다. 부모 프로세스의 메모리가 그대로 자식 프로세스의 메모리 구조로 사용되어 프로세스제어블록(PCB)도 PID를 제외하고는 동일하다. 프로그램 카운터가 동일하므로 다음에 수행될 위치는 프로그램의 fork() 다음 명령이 된다. 만약 자식 프로세스가 부모 프로세스와 다른 프로그램을 실행시키길 원한다면 exec()으로 원하는 프로그램을 실행시킬 수 있다. exec()을 사용하면 자식 프로세스는 복제된 메모리 구조와 프로세스 제어 블록의 내용을 새 프로그램의 내용으로 대체한다.

Windows는 CreateProcess() 시스템 호출을 통해 자식 프로세스를 생성한다. UNIX, Linux와 달리 부모 프로세스를 복제하지 않고 처음부터 새 프로그램으로 자식 프로세스를 생성한다. 하지만 자식 프로세스의 PID가 부모 프로세스에 전달되는 것은 동일.

부모 프로세스는 자식 프로세스의 PID를 이용해 자식 프로세스를 종료시킬 수 있다. 또 부모 프로세스가 종료되는 경우 OS에 의해 자식 프로세스가 모두 종료될 수 있다. 부모 프로세스는 살아있는데 자식 프로세스들이 종료된 경우 자식 프로세스의 자원은 OS에 의해 즉시 회수되지만 자식 프로세스의 프로세스 제어 블록은 부모 프로세스가 결과를 받을 때까지 사라지지 않고 유지된다.

**쓰레드:** 전통적 프로세스는 디스패칭 단위인 동시에 자원 소유의 단위. 하나의 메모리 구조(코드, 정적 데이터, 스택, 힙)와 코드 영역에 대한 하나의 제어흐름을 갖는다. 하나의 프로세스는 하나의 프로그램 카운터만을 가지게 됨으로 다중 처리가 불가능하다.

쓰레드는 프로세스 내에서의 다중처리를 위해 제안된 개념이다. 프로그램의 실행 단위를 프로세스보다 한 단계 낮추어 규정한 것이다. [전통 프로세스 -> 디스패칭 단위인 동시에 자원 소유의 단위], [현대 운영체제의 프로세스 -> 자원 소유의 단위, 쓰레드 -> 디스패칭 단위]. 결과적으로 하나의 프로세스 내에 여러 쓰레드를 두어 제어 흐름을 다중화했다. -> 프로세스에서 실행의 개념만 분리 -> 멀티쓰레드 프로세스

쓰레드는 실행에 필요한 최소의 정보만 가진다.(개별 프로그램 카운터를 포함한 레지스터 값과 스택 영역, 상태 정보) 나머지 정보는 한 프로세스 안에서 공유한다. 멀티쓰레드 프로세스는 멀티 코어 프로세서나 다중 프로세서를 지닌 환경에서 각 쓰레드마다 최적화된 연산을 병렬로 처리할 수 있다.

3장 프로세스 스케줄링

프로세스 스케줄링은 **공정성, 균형성을** 이루기 위해 동작한다. 공정성은 모든 프로세스에 적절한 수준으로 연산을 처리하는 것, 균형성은 시스템의 자원이 충분히 활용될 수 있게 하는 것이다.

공정성과 균형성을 이루기 위해서는 크게 네 가지 기준을 조절해야 한다.

**처리량:** 주어진 시간동안 처리한 프로세스 수

**반환시간:** 프로세스 생성부터 종료까지 걸린 시간

**응답시간:** 요청 시점부터 반응이 시작되는 시점까지 걸린 시간

**대기시간:** 프로세스가 종료될 때까지 준비 큐에서 기다린 시간의 합

**스케줄링 정책:** 스케줄링을 통한 결과에는 트레이드오프가 따른다. 따라서 스케줄링의 목표에 따라 다른 전략을 취해야 한다. 스케줄링 전략은 크게 선점/비선점 스케줄링으로 구분되며 이하 스케줄링에 대한 설명은 모두 하위 단계 스케줄링을 가정한다.

**선점 스케줄링:** 실행 중인 프로세스에 인터럽트를 걸고 다른 프로세스에 CPU를 할당할 수 있는 스케줄링 방식이다. 선점 방식은 높은 우선순위의 프로세스를 먼저 처리해야 하는 경우 유용하다.

**결과가 예측 가능해야 함, 시분할 시스템에서 빠른 응답시간을 유지해야 함** -> 선점 스케줄링이 적합

단점. 문맥 교환(Context Switching, 문맥 전환)이 필요하며 문맥 교환에 따른 오버헤드가 발생한다.

**문맥(Context):** CPU레지스터(**특히 프로그램 카운터**)와 기타 운영체제가 요구하는 프로세스의 상태

**문맥 교환:** CPU가 현재 실행 중인 프로세스의 문맥을 프로세스 제어 블록(PCB)에 저장하고 다른 프로세스의 프로세스 제어 블록으로부터 문맥을 복원하는 작업

**비선점 스케줄링:** 실행 중인 프로세스를 바로 준비상태로 전이시킬 수 없는 스케줄링 방식

강제 문맥 교환이 없기 때문에 오버헤드가 발생하지 않는다.

단점. 실행 시간이 짧은 프로세스가 오래 기다려야 될 수도 있다.

**스케줄링 알고리즘 평가 기준**

**평균대기시간, 평균 반환(대기시간+실행시간) 시간**

**선점 알고리즘:** FCFS, SRT, RR, 다단계 피드백 큐 스케줄링

**비선점 알고리즘:** SJF, HRN

**스케줄링 알고리즘**

**FCFS 스케줄링:** 준비 큐에 도착한 순서에 따라 프로세스 디스패치. 프로세스의 입력 순서에 따라 평균 반환 시간이 크게 증감한다. 실행 시간이 짧은 작업이 실행 시간이 긴 작업 사이에 끼어 있거나 뒤에 놓이면 비효율적인 알고리즘이다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 프로세스 | A | B | C | D |
| 도착시각 | 0 | 2 | 4 | 5 |
| CPU 사이클 | 7 | 4 | 1 | 3 |

평균대기시간 = (0 + 7 + 11 + 12)/4 = 7.5

평균반환시간 = (7 + 11 + 12 + 15)/4 = 11.25

만약 C D B A 순으로 프로세스가 놓인다면,

평균대기시간 = (0 + 1 + 4 + 8)/4 = 3.25

평균반환시간 = (1 + 4 + 8 + 15)/4 = 7

이 된다.

최근에는 여러 스케줄링 알고리즘을 조합해 최적화한다.

FCFS는 우선순위에 따라 프로세스를 스케줄링할 때 우선순위가 같으면 사용한다.

**SJF 스케줄링:** Shortest Job First. 우선 대기열에 있는 첫 번째 프로세스를 실행한다. 첫 프로세스를 실행하는 동안 준비 큐에 있는 프로세스 중 실행시간이 가장 짧은 것을 먼저 디스패치해 실행하는 비선점 방식의 알고리즘이다. 위 표를 SJF로 스케줄링하면 A-C-D-B 순서가 된다.

평균대기시간=(0 + 9 + 3 + 3)/4 = 3.75

평균반환시간=(7 + 13 + 4 + 6)/4 = 7.5

그러나 SJF는 실제 프로세스의 실행예정시간을 정확히 알 수 없어 사용자의 추정치의 의존한다. 비선점 방식이어서 새로 들어온 짧은 작업이 실행 중인 긴 작업을 기다리는 경우도 생겨 시분할 운영체제나 실시간 운영체제에는 적합하지 않다.

**SRT 스케줄링:** Shortest Remaining Time. 선점 방식의 SJF 알고리즘. 준비 큐에서 대기 중인 프로세스 중 남은 실행시간이 가장 짧다고 예상되는 것을 디스패치해 실행한다. 선점 방식이기에 새로 들어온 프로세스가 현재 실행중인 프로세스보다 예상실행시간이 짧으면 새로 들어온 프로세스가 즉시 디스패치된다. SJF와 마찬가지로 실행할 프로세스의 CPU 사이클이 미리 주어져야 적용 가능하다.

평균대기시간 = (8 + 1 + 0 + 2)/4 = 2.75

평균반환시간 = (15 + 5 + 1 + 5)/4 = 6.5

SJF보다 평균 대기/반환 시간이 짧지만 각 프로세스의 실행시간을 추적해 각 프로세스가 서비스를 받은 시간을 기록해야 한다. 또 선점을 위한 문맥교환 시에 오버헤드가 발생해 SJF보다 오버헤드가 크다.

**RR** **스케줄링:** Round Robin. 도착한 순서대로 프로세스를 디스패치. 정해진 시간 할당량(시간간격)에 따라 실행을 제한한다. 할당된 시간 안에 종료하지 못한 프로세스는 준비상태로 전이되어 큐의 마지막으로 간다.

적절한 시간 할당량을 사용하면 프로세스들이 CPU를 독점하지 않고 공평하게 이용할 수 있다. 시간 할당량이 0.1초이고 10개의 프로세스가 활성화되어 있다면 모든 프로세스는 1초 이내의 응답시간을 갖는다. 이런 방식은 시분할 운영체제에 적합하다.

**HRN 스케줄링:** Highest Response Ratio Next. 준비 큐에서 프로세스의 응답비율이 가장 큰 것을 먼저 디스패치해 실행하는 비선점 방식의 스케줄링 알고리즘. 응답비율 계산은 다음과 같다.

응답비율 = (대기시간 + 예상실행시간)/예상실행시간 = 대기시간/예상실행시간 + 1

예상실행시간이 짧을수록, 대기시간이 길수록 응답비율이 커진다. 모든 프로세스가 동시에 큐에 들어오면 그 중 예상실행시간이 가장 짧은 프로세스가 선택되고 예상시간이 모두 동일하지만 준비 큐에 들어온 시각이 모두 다르면 대기시간이 가장 긴 프로세스가 선택된다. HRN도 SJF나 SRT처럼 실행할 프로세스의 CPU 사이클이 미리 주어져야 적용 가능하다.

SJF의 단점을 보완한 스케줄링이다. 예상실행시간이 긴 프로세스가 준비 큐에 먼저 들어왔더라도 실행시간이 짧은 프로세스들에 의해 자꾸만 실행 순서가 밀리면 긴 프로세스의 대기 시간이 길어져 우선순위에서 밀리지 않을 수 있다.

**다단계 피드백 큐 스케줄링:** 입출력 중심 프로세스와 연산 중심의 프로세스를 구분해 서로 다른 시간 할당량을 부여하는 선점 방식의 스케줄링

**4장. 병행 프로세스1**

**병행성과 병행 프로세스:** 병행성은 여러 개의 프로세스가 동시에 실행되는 것. 병행 프로세스는 동시 실행 프로세스

병렬성은 크게 강결합(공유 메모리 구조)와 약결합(분산 메모리 구조)로 구현된다. 강결합은 여러 CPU가 하나의 기억장치를 공유(Master/Slave, SMP; Symmetric MultiProcessing)하지만 약결합은 2개 이상의 독립된 컴퓨터 시스템이 네트워크로 연결된다.(클러스터 구조) 각 시스템은 별도의 운영체제와 기억장치를 가진다. 시스템 간의 통신이나 메시지 전달을 위해서는 원격 프로시저 호출(RPC; Remote Procedure Calls)을 사용한다.

**프로세스 간의 관계:** 병행 프로세스는 프로세스 간의 관계에 따라 독립 프로세스와 협력 프로세스로 구분된다.

**독립 프로세스:** 수행 중인 다른 프로세스에 영향을 주지도, 받지도 않는다. 자신이 사용하는 데이터, 자신의 상태를 공유하지 않는다. 이러한 독립 프로세스의 실행은 결정적(Deterministic)이며 재생 가능하다. 이 말은 다른 프로세서의 영향을 받지 않기에 동일 인풋에 동일 아웃풋이 보장된다는 말이다. 다른 프로세스와 무관하게 중단하거나 재시작할 수 있다.

**협력 프로세스:** 독립 프로세스와 반대라고 생각하면 된다. 다른 프로세스와 영향을 주고 받으며 데이터, 상태를 공유한다. 협력 프로세스의 실행은 비결정적(Nondeterministic)이며 재생 불가능하다. 다른 프로세스와 영향을 주고 받기에 동일 인풋에 다른 아웃풋이 나올 수 있다. 일례로 은행의 입출금 시스템이 해당될 수 있다. A의 잔고에 5000원이 있는데 A가 B의 계좌로 10000원을 송금하려면 누군가가 A의 계좌로 5000원 이상의 돈을 입금해야 한다. 누군가가 A의 계좌로 5000원 이상 입금시 A는 B의 계좌로 10000원을 송금할 수 있고 그러지 않을 경우 송금할 수 없다.

**병행성 문제**

협력 프로세스는 병행성과 관련해 다음과 같은 문제를 야기할 수 있다.

**상호배제(Mutual Exclusion):** 2개 이상의 프로세스가 동시에 임계영역을 수행하지 못하도록 하는 것. 임계영역(Critical Section)은 2개 이상의 프로세스가 동시에 사용하면 안 되는 공유자원을 액세스하는 프로그램 코드 영역을 말한다.

잔고가 0원인 A의 계좌에 B는 5000원 C는 8000원을 입금하려 할 때 상호배제가 안 된 상태에서 둘이 동시에 입금하면 A의 계좌에는 5000원 또는 8000원만 입금되게 된다. 상호배제를 해 순차적으로 적용되어야 13000원이 입금될 수 있게 된다.

**동기화(Synchronization):** 2개 이상의 프로세스에 대한 처리순서를 결정하는 것. 10000원이 있는 계좌 A에서 10000원을 B 계좌로 송금하려면 A계좌에서 출금 -> B 계좌로 입금 순서를 따라야 한다. 이때 순서가 뒤바뀌어 B 계좌로 입금이 먼저 실행된 뒤 프로세스가 예기치 않게 종료되면 A와 B의 계좌 모두에 10000원이 남게 된다. 위의 상호배제도 임계영역에 대한 동기화 문제로 볼 수 있다.

**프로세스 간 통신(IPC; Inter-Process Communication):** 협력 프로세스에서 동일 변수를 공유하거나 메시지를 주고받는 데에 IPC는 필수이다. 통신 방법에 따라 상호배제와 동기화 문제를 포함 다양한 문제가 발생한다. 구체적인 내용은 5장에서 다룬다.

**세마포어**

Semaphore는 상호배제와 동기화 문제를 해결하기 위해 다익스트라(Dijkstra)가 제안한 도구

**세마포어의 정의: (**관례적)변수 s로 선언, 사용 가능한 자원의 수 또는 잠김, 풀림의 상태를 값으로 갖는다. 선언 시 상황에 맞는 값(0이상 정수)으로 초기화한다. 이후 세마포어는 두 개의 기본연산(primitive operation) P와 V를 통해서만 사용될 수 있다. P와 V를 통하지 않고서는 s의 값을 확인하거나 수정할 수 없다.

연산 P는 네덜란드어로 검사(Proberen) 또는 감소시키려는 시도(Probeer Teverlagen)을 나타내며 down, wait등의 이름으로 사용하기도 한다. 연산 V는 네덜란드어로 증가(verhogen)를 나타내며, Up, Signal 등의 이름으로 사용되기도 한다.

void P(semaphore s) {

if(s > 0)

s--;

else

대기 큐에 현재 프로세스 대기;

}

void V(semaphore s) {

if(대기 큐에 대기 중인 프로세스 없음)

s++;

else

대기 중인 프로세스 1개 준비상태로 전이시킴(FIFO 큐 가정);

}

연산 P와 V의 정의는 위와 같다.

**상호배제 해결:** 세마포어로 임계영역에 대한 상호배제 문제를 해결할 수 있다. 우선 임계 영역의 코드 영역을 진입 영역과 해제 영역의 코드로 감싼다. 진입 영역과 해제 영역에는 각자 프로세스가 임계영역 수행을 시작할 수 있는지, 끝났는지 체크하는 코드를 둔다.

추가적으로 진입영역에서 임계영역으로 넘어가지 못하고 대기하는 프로세스들은 적절한 시간 내에 임계영역 수행을 할 수 있어야 한다. 적절한 시간이란 어떤 프로세스가 자기보다 먼저 진입영역에 도착한 프로세스들이 모두 임계영역을 수행하는 만큼의 시간을 크게 벗어나지 않는 수준의 시간이다.

세마포어를 이용해 상호배제를 해결하는 방법은 진입 영역에 연산 P를, 해제 영역에 연산 V를 둠으로써 이뤄진다. 이때 세마포어 mutex의 초깃값을 1로 둔다.

초깃값 1은 어떤 프로세스도 임계영역을 수행하고 있지 않음을 뜻한다. 한 프로세스가 임계영역을 실행하게 되면 초깃값은 0으로 세팅되어 다른 프로세스가 대기 큐에서 임계영역을 실행할 수 없도록 한다. 한 프로세스가 임계 영역의 실행을 마친 뒤 대기 큐에 아무런 프로세스가 없으면 mutex값은 다시 1로 초기화되고 다른 프로세스가 있다면 mutex값은 여전히 0인 상태에서 가장 오래 기다린 프로세스를 준비 상태로 전이시킨다. 준비상태로 전이된 프로세스가 디스패치되어 실행상태가 되면 연산 P가 완료되며, 임계영역 수행을 시작하게 된다.

**동기화 해결:** 프로세스 A가 코드 S1을 수행하고 프로세스 B가 코드 S2를 수행하도록 동기화하는 상황을 가정한다. 이 경우 코드 S1 뒤에 연산 V를 두고 코드 S2 앞에 P를 둔다. 이때 세마포어 sync의 초깃값은 0이다.

Sync는 나중에 실행되는 S2의 수행 시작 가능 여부를 나타낸다. 초깃값 0은 실행 불가를 의미하며 이는 코드 S1이 아직 수행되지 않았기에 S2가 실행될 수 없음을 의미한다.

코드 S1 뒤에 V를 두면 프로세스 A가 S1 수행을 마치고 V(sync)를 추가할 때 sync의 대기 큐가 비어 있다면 sync만 1로 바뀐다. 만약 대기 큐가 비어 있지 않다면 대기 중인 프로세스 B를 큐에서 준비상태로 전이시킨다. 전이된 프로세스 B가 디스패치되어 실행상태가 되면 연산 P가 완료되며 코드 S2수행을 시작하게 된다.

**5장. 병행 프로세스2**

**생산자-소비자 문제**

생산자-소비자로 이뤄진 두 협력 프로세스 사이에 버퍼를 두고 생산자 프로세스에서는 데이터를 넣고 소비자 프로세스에서는 데이터를 꺼낼 때 상황을 다루는 문제이다.

유한 버퍼 문제(Bounded Buffer)의 조건

1. 버퍼에 여러 프로세스가 동시에 접근할 수 없다.
2. 버퍼의 크기는 유한하다.

첫 번째 조건은 상호배제가 필요하다는 것을 의미 -> 데이터를 넣는 동시에 인출할 수 없다.

두 번째 조건은 동기화가 필요하다는 것을 의미 -> 버퍼가 가득 차거나 빌 경우 대기해야 한다.

**세마포어를 이용한 해결**

**생산자 코드**

While(true) {

데이터 생산

P(empty);

P(mutex);

버퍼에 데이터 넣음

V(mutex);

V(full);

}

**소비자 코드**

While(ture){

P(full);

P(mutex);

버퍼에서 데이터 꺼냄

V(mutex);

V(empty);

데이터 소비

}

세마포어 mutex = 1, empty = n(버퍼의 크기), full = 0으로 초기화한다.

생산자가 데이터를 넣고 소비자가 데이터를 꺼내는 부분은 동시에 수행될 수 없으므로 임계영역이다. 따라서 임계영역 앞에 P(mutex)를 두고 임계영역 뒤에 V(mutex)를 두어 상호 배제를 해결한다.

다음으로 버퍼가 가득 찬 경우의 동기화를 위해 세마포어 emptry를 사용한다.

Empty는 버퍼에 존재하는 빈 공간의 개수이다. N으로 초기화하고 버퍼가 가득차면 0이 된다.

세마포어 empty가 0인 상황에서 생산자는 데이터를 생산하더라도 소비자가 버퍼에서 데이터를 꺼낸 후에야 버퍼에 데이터를 넣을 수 있다. 따라서 생산자가 버퍼에 데이터를 넣는 코드 앞에 P(empty)를 두고 소비자는 버퍼에서 데이터를 꺼내는 코드 뒤에 V(empty)를 둬 버퍼가 가득 찬 경우 동기화를 해결해야 한다.

한편, 버퍼가 빈 경우는 동기화를 위해 세마포어 full을 이용한다. Full은 버퍼에 존재하는 데이터의 개수이다. Full이 0인 상황에서 소비자는 사용할 데이터가 없으므로 생산자가 버퍼에 데이터를 넣은 후에야 데이터를 꺼낼 수 있다. 따라서 소비자는 버퍼에서 데이터를 꺼내는 코드 앞에 P(full)d을 두고 생산자는 버퍼에 데이터를 넣는 코드 뒤에 V(full)을 두어 버퍼가 빈 경우 동기화를 해결한다.

**판독기-기록기 문제**

협력 프로세스들이 파일같은 공유 자원을 두고 데이터를 쓰거나(기록기) 읽는(판독기) 상황을 다루는 문제이다.

조건

1. 하나의 기록기가 공유 자원에 데이터를 쓰는 중에 다른 기록기나 판독기는 공유 자원에 접근할 수 없다.
2. 여러 판독기는 동시에 공유자원에서 데이터를 읽을 수 있다. 판독기가 데이터를 읽는 중에 기록기는 공유자원에 접근할 수 없다.

첫 번째 조건 – 상호 배제로 해결

두 번째 조건 – 판독기가 읽는 중에 기록기가 자원에 접근해 대기 중이다. 이때 새 판독기가 공유 자원에 접근한다면 두 번째 조건에 의해 판독기는 바로 접근해야 할까? 아니면 이미 대기 중인 기록기가 있으므로 대기해야 할까? 다음 두 문제로 정의할 수 있다.

1. **제1판독기-기록기 문제**

기록기보다 판독기에 우선순위를 준다. 기록기의 대기 여부에 관계없이 새로운 판독기는 우선적으로 공유자원에 접근할 수 있다. 판독기-기록기 문제의 두 번째 조건을 충실히 따르지만 기록기가 기아 상태에 빠질 수 있다는 단점이 있다. 기아 상태는 프로세스가 필요한 자원을 할당받지 못하고 계속 대기하는 상황이다. 판독기가 공유 자원에 접근 중인 상황에서 새 판독기가 지속적으로 등장하면 기록기는 언제까지고 대기에 머물다가 기아상태에 빠진다.

1. **제2판독기-기록기 문제**

1과 달리 판독기보다 기록기에 우선 순위를 준다. 기록기가 기아 상태에 빠지는 걸 방지하지만 두 번째 조건에 위배되고 판독기의 병행성이 떨어진다. 또 기록기에 우선순위를 주는 방식에 따라 판독기가 기아상태에 빠질 수 있다.

**세마포어를 이용한 해결:** 제1판독기-기록기 문제는 세마포어를 이용해 해결 가능하다.

**2개의 세마포어를 이용한 제1판독기-기록기 코드(wrt = 1, mutex = 1, rcount = 0)**

**기록기 코드**

P(wrt);

공유자원에 쓰기

V(wrt);

**판독기 코드**

P(mutex)

Rcount = rcount + 1;

If(rcount == 1) then P(wrt);

V(mutex);

공유자원에서 읽기

P(mutex);

Rcount = rcount -1;

If(rcount == 0) then V(wrt);

V(mutex);

기록기에 대한 상호배제를 위해 세마포어 wrt(write)를 이용한다. 기록기가 쓰는 부분은 임계영역이 되고 따라서 임계영역 앞 뒤에 P(wrt), V(wrt)를 두어 상호배제를 해결한다.

다음으로 판독기가 공유자원에서 데이터를 읽는 부분은 기록기가 쓰는 부분과 동시에 수행될 수 없으므로 역시 임계영역이 된다. 기록기처럼 P(wrt)와 V(wrt)를 앞뒤에 둔다. 그런데 여러 판독기는 동시에 공유자원의 데이터를 읽을 수 있으므로 하나의 판독기가 임계영역을 수행 중일 때도 다른 판독기가 임계영역 수행을 시작할 수 있어야 한다. rcount는 이를 위해 존재한다. rcount는 동시에 공유자원의 데이터를 읽는 판독기의 개수를 센다. rcount가 1보다 크면 공유자원을 읽고 있는 다른 판독기가 이미 있다는 뜻이므로 상호배제가 되지 않도록 P(wrt)를 수행하지 않고 바로 임계영역 수행을 시작한다. 마찬가지로 임계영역 수행이 끝나 rcount를 1만큼 줄였을 때 여전히 0보다 크면 공유자원을 읽고 있는 다른 판독기가 아직 있다는 말이므로 V(wrt)를 수행하지 않도록 한다. 변수 rcount의 값을 변화시키고 확인하는 작업은 다른 판독기에 의해 방해받지 않고 일괄적으로 처리되어야 하므로 이 또한 임계영역으로 본다. 따라서 세마포어 mutex를 이용해 상호배제가 되도록 해야한다. 더불어 기록기가 임계영역을 수행 중이어서 P(wrt)로 대기 중인 판독기가 있을 때, 추가적인 판독기들은 P(mutex)에 의해 대기 가능하다.

제2판독기-기록기 문제 또한 세마포어를 통해 해결 가능하다.

**rd, wrt, mutex1, mutex2, mutex3 =1**

**rcount, wcount = 0**

**기록기 코드**

P(mutex2);

wcount = wcount + 1;

if(wcount == 1) then P(rd);

V(mutex2);

P(wrt);

공유자원에 쓰기

V(wrt);

P(mutex2);

wcount = wcount – 1;

if(wcount == 0) then V(rd);

V(mutex)2;

**판독기 코드**

P(mutex3);

P(rd);

P(mutex1);

rcount = rcount + 1;

if(rcount == 1) then P(wrt);

V(mutex1);

V(rd);

V(mutex3);

공유자원에서 읽기

P(mutex1);

rcount = rcount – 1

if(rcount == 0) then V(wrt);

V(mutex1);

기록기가 대기 중일 때 이후에 새로운 판독기가 들어오면 임계영역을 수행하지 못하도록 세마포어 rd가 제1판독기-기록기 코드에 추가되었다. 또한 기록기에 우선순위를 주기 위해 mutex3e도 추가되었다.

**프로세스 간 통신(IPC; InterProcess Communication)**

공유 메모리를 이용하거나 메시지를 이용한 방식이 있다. 두 가지 방식이 한 운영체제 안에서 함께 사용될 수 있다.

**공유 메모리 방법**

협력 프로세스가 모두 동일한 변수를 사용해 데이터를 서로 공유하는 방법

해당 변수가 사용하는 메모리는 협력 프로세스들이 모두 접근 가능한 공유 자원이다. 공유 메모리 방법의 예는 생산자-소비자 문제의 유한 버퍼와 판독기-기록기 문제의 공유자원이 해당한다.

각 프로세스가 메모리에 직접 접근해 대량의 데이터를 쓰거나 읽을 수 있어 빠른 속도로 통신할 수 있다. 그러나 상호 배제나 동기화 등 발생할 수 있는 모든 문제를 응용 프로그래머가 책임지고 해결해야 한다.

**메시지 전달방법**

협력 프로세스들이 커널(연산 send, receive)을 통해 메시지를 주고 받으며 데이터를 공유한다. 매번 시스템 호출을 통해 메시지를 전달해야 하므로 대량의 데이터보다 소량의 데이터 송수신에 유리한 방법이다. 통신상에 발생할 수 있는 문제는 운영체제가 책임진다. 다음은 메시지의 전달방법과 논리적인 구조이다.

1. **통신 링크:** 두 프로세스가 메시지를 주고 받을 때 두 프로세스 사이에는 통신 링크가 있다. 이때 두 프로세스 사이에는 오직 하나의 링크만 존재할 수도, 여러 개의 링크가 존재할 수도 있다. 방향성도 양방향-단반향으로 설정 가능, 링크의 용량을 여러 형태로 설정도 가능. 통신 링크의 용량은 링크가 자체적으로 갖는 큐(버퍼)에 보관할 수 있는 메시지의 수를 의미한다. 큐의 크기가 0 이면 송신자는 수신자가 메시지를 받을 때까지 기다리고 무한이면 송신자는 아무 때나 메시지를 보낼 수 있다. 유한한 경우 송신자는 버퍼가 비워질 때까지 송신을 대기해야 한다.
2. **직접통신:** 두 프로세스가 서로를 직접 지정하여 메시지를 주고받는다. 송신자는 send 연산에 수신자를 직접 명시하고 수신자는 연산 receive에 송신자를 직접 명시한다. 프로세스 A가 B에게 메시지를 보낼 때 -> send(B, M) -> receive(A, M)

직접 통신은 두 프로세스 사이에 오직 하나의 통신 링크가 자동으로 설정된다. 또한 두 프로세스가 송수신을 바꿔서 할 수 있도록 링크는 양방향이다. 송신자와 수신자가 각각 상대를 직접 명시하는 경우 이는 대칭형 주소 지정이다. 수신자가 송신자를 지정하지 않고 메시지를 받을 때 송신자의 이름을 같이 받는 경우를 비대칭형 주소 지정이라고 한다. 비대칭형은 receive(id, m)처럼 사용한다. 송신자의 ID를 id에, 메시지를 m에 받는 형태이다.

수신자가 다수의 송신자와 통신 링크를 갖는 경우 사용할 수 있다.

1. **간접통신:** 프로세스 사이에 우편함(Mailbox)를 두고 이를 통해 메시지를 주고받는다. 송신자는 send에 수신자 대신 우편함을 명시하고 수신자도 receive 대신 우편함을 명시한다.

두 프로세스 간 통신 링크는 두 프로세스가 같은 우편함을 이용하는 경우 설정된다. 만약 두 프로세스가 여러 개의 우편함을 이용해 메시지를 주고 받으면 여러 개의 통신 링크가 존재하게 된다. 여러 프로세스가 하나의 우편함을 이용하는 경우 하나의 링크가 여러 프로세스와 연관되게 된다.

**6장. 교착상태1**

**프로세스의 자원 할당 절차:** 프로세스는 일반적으로 요구 -> 사용 -> 해제(반납)의 자원 사용 절차를 가진다. 프로세스 1이 자원을 사용 중일 때 프로세스 2는 요구 상태에서 대기하다가 프로세서 1이 자원을 해제하는 순간 해당 자원을 획득해 사용하게 된다.

**Deadlock(교착상태):** 여러 개의 프로세스가 위와 같은 자원 할당 절차 속에서 서로 상대 프로세스의 작업이 끝나기만 기다리는 상태를 교착상태라고 한다. 예를 들어 위 상황에서 프로세스 1이 자원 1을 획득한 상태에서 자원 2를 요구하고 프로세스 2는 자원 2을 획득한 상태에서 자원 1을 요구한다면 두 프로세스는 서로 상대방이 작업을 끝내고 자원을 해제하기만 기다리는 교착상태가 된다. 교착상태는 이처럼 관련된 모든 프로세스가 자원 획득의 가능성 없이 무한히 대기상태인 것을 의미하므로, 자원 획득의 가능성은 있으나 지속적으로 대기상태인 기아상태(Starvation)와는 다르다.

**교착상태의 필요조건**

1. **상호배제 조건(Mutual Exclusion)**

적어도 하나 이상의 자원이 동시에 사용될 수 없다. 프로세스가 자원에 대한 배타적 통제권을 요구하는 조건으로 둘 이상의 프로세스가 한 자원에 접근할 수 없게 된다.

1. **점유대기 조건(Hold and Wait)**

프로세스가 이미 한 자원을 할당받아 점유 중인 상황에서 다른 프로세스가 점유하고 있는 다른 자원을 요청하여 해제되기를 기다리고 있는 상황이다.

1. **비선점 조건(No Preemption)**

할당된 자원은 타의에 의해 해제되지 않는다.

1. **환형대기 조건(Circular Wait)**

점유된 자원과 자원요구 관계가 환형 관계를 이루며 대기하는 상황. A -> B -> C -> A

**자원할당 그래프**

**G=(V,E)**는 방향 그래프로 교착상태를 명확하게 표현하기 위해 이용하는 그래프이다.

**V:** 정점의 집합. V = PR

* P = {}: n개의 프로세스
* R = {}: m개의 자원

**E:** 방향 있는 간선의 집합. E = QS

* Q = {( | }: 프로세스 가 자원 를 요구한다(요구간선)
* S = {() | , }: 자원 가 프로세스 에 할당된다(할당간선)

각 프로세스 는 원으로, 자원 는 사각형으로 나타낸다. 자원은 프린터나 디스크처럼 자원의 유형을 나타내기에 각 자원 마다 하나 이상의 단위자원을 가질 수 있다. 따라서 각 자원 사각형 위에 단위자원의 개수를 숫자로 표시한다.

**상호배제:** 하나의 할당간선으로 표현

**점유대기:** 한 프로세스에 할당간선과 요구간선이 연결된 것으로 표현

**비선점:** 가용한 단위자원이 없는 경우 요구간선으로 표현

**환형대기:** 할당간선과 요구간선으로 만들어지는 사이클

사이클이 형성되면 교착상태 발생 가능, 사이클 없으면 교착상태 x

또 교착상태가 되었다가 가용한 리소스가 생겨 해제될 수도 있다.

**교착상태 처리기법**

1. **예방(Prevention):** 교착상태의 네 필요조건이 동시에 만족되는 것을 피한다.
2. **회피(Avoidance):** 프로세스에 필요한 자원의 최대량에 대한 정보를 이용해 교착상태가 발생하지 않도록 한다.
3. **탐지 및 복구(Detection and Recovery):** 교착상태 발생 여부를 조사해 발생한 경우 적절한 조치를 취해 정상상태로 복구한다.

**교착상태 예방:** 교착상태가 발생하려면 네 가지 필요조건이 모두 충족되어야 한다. 각 필요조건의 제거가 가능한지 확인하고, 가능하다면 제거해 교착상태를 예방한다.

1. **상호배제 조건 제거:** 읽기 전용 파일의 경우 여러 프로세스의 동시 접근이 가능하지만 그 외의 공유 불가능한 자원에 상호배제 조건을 제거하는 것은 불가능하다.
2. **점유대기 조건 제거:** 대기할 때 자원을 점유해서는 안 되고 자원을 점유할 때 대기해서는 안 되게 한다. 프로세스가 필요한 자원을 한꺼번에 모두 할당 받도록 하면 점유대기 조건을 제거할 수 있다. 하지만 모든 자원을 미리 점유해 실제 자원이 필요한 시점까지 대기하는 자원이 생겨 자원 이용률이 낮아진다. 또 요구한 자원 중 하나라도 바로 할당받을 수 없는 경우 대기해야 하므로 기아상태에 빠질 수 있다.

대기할 때 자원을 점유할 수 없도록 하려면 새 자원을 요구할 때 먼저 할당한 자원을 모두 해제하면 된다. 할당한 자원이 없으므로 점유대기 조건을 제거할 수 있다. 하지만 프린터처럼 점유 도중 해제할 수 없는 자원에는 적용할 수 없다.

1. **비선점 조건 제거:** 프린터 같은 자원의 경우 작업 도중 선점하기는 힘들다. 차선책으로 자원을 점유한 프로세스가 다른 자원을 요구할 때 대기가 발생하면 할당했던 자원을 모두 해제해 다른 프로세스가 비선점 조건으로 대기할 가능성을 줄일 수 있다. 하지만 이 또한 프린터에는 적용할 수 없다.
2. **환형대기 조건 제거:** 모든 m개의 자원 에 서로 다른 자연수로 번호를 매기고 이를 지정하는 함수 f를 정의한다. R\_i != R\_j 이면 f(r\_i) != f(r\_j)이다. 이와 같은 방식으로 프로세스가 자원을 요구할 때 항상 오름차순의 자원을 할당하면 환형대기 조건을 제거할 수 있다. 또 프로세스가 특정 자원을 요구 시 해당 자원보다 일련번호가 큰 자원을 해제하도록 해도 환형대기 조건을 제거할 수 있다. 자원번호의 배정이 어렵다는 문제가 있다.

**7장. 교착상태2**

**교착상태 회피:** 자원 사용에 대한 사전 정보를 파악해 교착상태가 발생하지 않도록 하는 방법. 현재 할당된 자원, 가용 상태의 자원, 프로세스의 최대 요구량 등이 자원 사용에 대한 사전 정보가 된다.

**안전상태와 안전순서열: 안전상태란** 교착상태를 회피하면서 각 프로세스에 그들의 최대 요구량까지 빠짐없이 자원을 할당할 수 있는 상태를 말한다. 안전상태는 안전순서열을 전제로 한다. 안전순서열은 순서가 있는 프로세스의 집합으로 각 프로세스 는 가 추가로 요구할 수 있는 자원의 양이 현재 가용상태의 자원으로 충당되거나 현재 가용상태인 자원에 i > j인 프로세스 에 할당된 자원까지 포함하여 충당 가능한 경우를 말한다. 이는 프로세스가 당장 자원을 할당받지 못해도 안전순서열에서 자신보다 순서가 빠른 프로세스들이 수행 후 자원을 해제할 때까지 기다리면 결국 자원을 얻을 수 있기 때문에 교착상태에 빠지지 않고 모든 프로세스가 작업을 마칠 수 있음을 의미한다.

**불안전상태:** 안전순서열이 존재하지 않는 경우를 말한다. 안전상태, 불안전상태, 교착상태의 관계는 아래와 같다.

교착상태

불안전 상태

안전 상태

**교착상태 회피 방법**

1. **각 자원의 단위자원이 하나밖에 없을 경우**

자원할당 그래프를 변형해 자원 정점에 표기하던 단위 자원을 없애고 선언간선을 추가한다. 선언간선()은 점선으로 표시되며 프로세스 가 언제일지는 모르지만 앞으로 분명히 자원 를 요구할 것임을 나타낸다. 프로세스가 자원을 요구할 때 선언간선은 요구간선인 실선으로 바뀐다. 요구간선은 다시 할당간선으로 바뀐 후 자원이 프로세스에서 해제될 때 선언간선으로 바뀐다. 삭제 대신 선언 간선이 되는 이유는 동일 프로세스가 동일 자원을 다시 요구할 수도 있기 때문이다.

1. **각 자원의 단위자원이 여러 개일 경우**

각 자원의 단위자원이 여러 개일 수 있는 일반적인 경우 은행원 알고리즘을 통해 교착상태를 회피한다. 은행원 알고리즘은 자원을 요구 받으면 자원을 할당해주고 난 후의 상태를 여러 데이터를 이용해 계산해서 그것이 안전상태인지 확인한 후 안전상태가 보장되는 경우에만 자원을 할당한다. 할당 이후의 상태 계산을 위해 은행원 알고리즘은 다음과 같이 상태 데이터를 관리한다. 자원의 종류는 m개로 가정한다.

프로세스의 p\_i의 각 자원에 대한 최대 요구량을 나타낸다. Mx\_ij는 프로세스 p\_i가 최대로 요구하는 자원 r\_j의 단위자원 개수이다.

프로세스 p\_i의 현재 할당량을 나타낸다. Alc\_ij는 현재 프로세스 P\_i에 할당된 자원r\_j의 단위자원 개수이다.

프로세스 p\_i의 각 자원에 대한 추가 요구량을 나타낸다. Nd\_ij는 앞으로 프로세스 p\_i가 추가로 요구할 수 있는 자원 r\_j의 단위자원 개수이다. 항상 가 성립한다.

현재 가용한 각 자원의 단위자원 수를 나타낸다. Av\_j는 자원 r\_j의 현재 가용한 단위자원 개수이다.

는 프로세스 p\_i가 1 <= j <= m인 모든 j에 대해 자원 r\_j를 k\_j개 요구하는 상황을 나타낸다. 프로세스 p\_i의 자원 요구가 발생하면 다음 은행원 알고리즘이 교착상태를 피하는 경우에만 자원을 할당한다.

텍스트, 책, 종이, 인쇄이(가) 표시된 사진

자동 생성된 설명

상태 데이터 사이에 부등호 <= 관계는 m개의 값 모두에 대해 <=가 성립함을 의미.

REQ <= NEED가 아닌 경우는 프로세스 p\_i가 추가로 요구할 수 있는 양보다 더 많은 요구가 발생한 자원이 있는 경우이므로 오류로 판단한다. REQ <= AVAIL이 아닌 경우는 현재 가용한 자원보다 더 많은 요구가 발생한 자원이 있는 겨우이므로 대기한다. REQ <= NEED이고 REQ <= AVAIL이면 오류도 없고 가용한 자원도 있는 상태이므로 안전상태가 유지되는지 확인한다. 이를 위해 상태 데이터를 할당 후의 상태로 만든다. REQ만큼 가용한 자원을 줄이고 할당량을 늘리며 추가 요구량을 줄인다. 실제 할당은 일어나지 않고 상태 데이터만 변화시킨다.

그 후 안전상태 확인을 위해 안전 알고리즘 safe()를 호출한다. 안전 알고리즘은 안전순서열이 존재하는지 유무로 안전상태를 파악한다. 아래 알고리즘의 결과 할당 후에도 안전 상태가 유지되면 REQ대로 할당하고 안전상태가 유지되지 않으면 프로세스는 대기시키고 할당 후의 상태로 바꾸었던 상태 데이터도 원래대로 복구시킨다.

텍스트, 책, 종이, 문서이(가) 표시된 사진

자동 생성된 설명

**교착상태 탐지 및 복구**

**탐지:** Shoshani와 Coffman 알고리즘은 은행원 알고리즘과 유사하지만 현재 상태의 모든 자원 요구량을 고려해 교착상태 여부를 파악한다.

**복구:** 교착상태가 탐지되면 복구 조치에 들어간다. 복구의 주체는 오퍼레이터와 운영체제 두 가지가 있다. 오퍼레이터에게 교착상태가 발생했음을 알려 오퍼레이터가 직접 처리하게 하는 것이 첫 번째 방법이다. 두 번째로 운영체제가 자동으로 교착상태로부터 복구하게 할 수도 있다.

**복구 방법:** 교착상태 프로세스 종료(전부 일시 종료, 사이클이 제거될 때까지 하나씩 종료), 교착상태 프로세스 자원 해제(환형대기 제거)

**8장. 메모리 관리**

**속도 -> 보조기억장치 < 메모리 < 캐시 메모리 < CPU레지스터**

**단일 프로그래밍 환경:** 하나의 연속된 블록으로 할당되는 연속 메모리 할당 방식을 이용했다. 메모리 용량보다 큰 프로세스를 실행할 수 없고 사용하지 않는 프로세스도 메모리에 계속 적재되어야 한다는 단점이 있다. 또 CPU 작업 중에는 입출력 장치가 대기하고 입출력 수행 중에는 CPU가 대기해야 하는 환경이다.

**다중 프로그래밍 환경:** 멀티프로그래밍이라고도 한다. 현재 프로세스가 입출력 대기 상태라면 다른 프로세스에 CPU를 할당할 수 있다. CPU연산과 입출력을 동시에 해서 CPU 이용도와 시스템 처리량을 증가시킬 수 있다.

**메모리 분할: 고정 분할과 동적 분할로 구분.** 여러 프로세스를 메모리에 적재하기 위한 방법으로하나의 분할에 하나의 프로세스가 적재된다.

**고정 분할:** 프로세스를 고정된 영역에만 할당하는 방식. 첫 번째로 고정된 물리 메모리 주소를 지정해 프로세스를 할당하는 방식이 있다. 컴파일 시에 프로그램 내의 주소를 절대주소로 번역해 특정 분할 영역에만 적재하는 방식이다. 큐가 빈 분할이 있어도 다른 프로세스를 적재할 수 없어 효율성이 떨어진다. 두 번째 방법은 메모리 전체에 하나의 큐만 두고 모든 프로세스를 하나의 작업 큐에 넣어 어느 분할에서든지 실행 가능하게 하는 것이다. 컴파일 시에 프로그램 내의 주소를 상대주소로 번역해 어디든 적재가 가능하다. 작업의 주소 지정 문제는 재배치 컴파일러, 어셈블러, 로더 등으로 해결한다. 절대번역 및 적재보다 메모리를 효율적으로 사용할 수 있지만 재배치 기능 번역기와 로더는 절대번역기보다 복잡하다.

**내부단편화(Fragmentation):** 고정 분할 시 프로세스의 크기가 분할된 영역보다 작아서 생기는 메모리 낭비.

**동적 분할:** 각 프로세스에 필요한 만큼만 메모리를 할당한다. 따라서 내부 단편화가 발생하지 않는 대신 외부 단편화가 발생한다.

**외부단편화(External Fragmentation):** 메모리의 할당과 반환이 반복됨에 따라 작은 크기의 공백이 메모리 공간에 흩어져 생기는 것. 해결 방법으로 통합(Coalescing)과 집약(Compaction)이 있다. 통합은 한 작업이 끝났을 때 해당 프로세스가 다른 비어있는 메모리와 인접해있는지 파악해 인접해 있다면 반환된 메모리 공간을 인접한 빈 공간과 통합하는 것이다. 집약은 메모리 내의 공간을 한 곳으로 모으는 작업이다. 여러 개의 작은 공백을 집약해 하나의 커다란 저장공간으로 만든다.

**메모리 보호:** 연속 메모리 할당 방식은 메모리 침범의 여지가 있다. 따라서 프로세스가 사용할 수 있는 주소 범위를 하한-상한 레지스터 쌍 또는 하한-크기 레지스터 쌍의 값으로 제한해 할당영역을 침범하지 않게 한다. 프로세스가 이 제한을 넘어 운영체제를 호출하려면 시스템 호출을 해야 한다.

**메모리 배치기법:** 동적 분할 다중 프로그래밍에서 새로 반입된 프로그램이나 데이터를 메모리의 어디에 배치시킬지 결정하는 것이다. 운영체제는 빈 공간 리스트를 가진다. 여기서 적합한 공간을 찾으면 된다. 네 가지 배치기법이 있다.

1. **최초 적합:** 빈 공간 리스트를 메모리 주소순으로 유지한다. 프로세스를 가장 먼저 발견된 빈 공간에 적재한다.
2. **후속 적합:** 최초 적합이 끝나고 난 다음 메모리부터 다시 최초 적합을 적용한다.
3. **최적 적합:** 필요한 메모리 공간 중 가장 작은 공간에 프로세스를 적재한다.
4. **최악 적합:** 메모리 공간이 가장 큰 곳을 찾아 프로세스를 적재한다. 메모리가 너무 작은 단위로 내부단편화되어 사용할 수 없게 되는 현상을 줄일 수 있다.

**9장. 가상 메모리**

**가상메모리:** 메모리보다 큰 프로세스를 실행하기 위해 사용하는 방식이다. 연속 메모리 할당 방식으로는 실행이 불가능하다. 실행 프로세스가 참조하는 주소를 가상 주소라고 하고 실제 메모리에서 사용하는 주소를 실주소 또는 물리적 주소로 분리한다. 전체 프로그램에서 프로세스가 필요로 하는 일부만 메모리에 적재함으로써 프로세스 실행을 가능하게 한다. 이때 실행 프로세스가 참조하는 가상주소의 범위를 프로세스의 가상주소 공간(Virtual Address Space) V라고 하며 특정 컴퓨터 시스템에서 사용 가능한 실주소의 범위를 실주소 공간(Real Address Space) R이라고 한다.

**사상,매핑(Mapping):** 프로세스는 오직 가상 주소만을 참조한다. 그리고 가상 주소는 실제 메모리에서 실행되는 실주소로 변환된다. 이 변환과정을 사상 또는 매핑이라고 하며 이는 변환함수로 표시된다. 이 작업은 운영체제가 처리한다. 사상작업 또한 연산 능력을 필요로 해서 이를 빨리 하기 위한 다양한 방법들이 개발되었다.

**동적주소변환(Dynamic Address Translation: DAT):** 가상주소를 실주소로 바꾸는 절차. 가상 주소상의 연속성이 실주소에서도 이어져야 하는 것은 아니다. 이러한 특징을 인위적 연속성이라고 한다. 사용자는 프로그램과 데이터가 실제 메모리 상의 어디에 위치해야 하는지 알 필요가 전혀 없다. 동적주소변환 방식은 가상 메모리와 현재 메모리 주소가 표시된 주소변환 사상표(Mapping Table)을 가지는데 이게 너무 자세하면(다루는 용량이 작아질수록) 메모리를 많이 차지하기에 적절한 트레이드 오프가 필요하다. 일반적인 방식은 블록 단위로 주소변환하는 블록 사상 시스템을 사용한다.

**블록 단위 주소변환:** 블록의 시작주소 + 변위(Displacement)면 블록이 어디서 시작해 어디서 끝나는지 알 수 있다. 이를 가상주소 v는 순서쌍(시작주소, 변위)이라고 표기한다. 블록의 크기가 커질수록 전송시간이 늘어나고 한 블록이 차지하는 메모리가 커져서 메모리를 공유할 수 있는 프로세스의 수가 줄어든다. 블록의 크기가 작아지면 전송시간이 줄어들고 메모리를 공유할 수 있는 프로세스의 수는 늘어나는 대신 사상정보에 필요한 메모리 양이 커진다.

**페이지:** 블록의 크기가 동일할 때 블록들을 페이지라고 부른다. 이런 가상 메모리 구성 기법을 페이징 기법이라고 한다.

**세그먼트:** 페이지와 달리 크기가 서로 다를 수 있는 블록이다. 이런 가상 메모리 구성 기법을 세그멘테이션 기법이라고 한다. 페이지와 결합하여 고정된 크기의 페이지로 구성된 다양한 크기의 세그먼트를 구현하는 시스템도 있다.

**페이징 기법:** 메모리를 고정된 크기의 블록인 페이지 단위로 나누어 관리하는 기법. v = (start, displacement). 직접사상과 연관사상, 직접/연관사상이 있다.

**직접사상:** 직접 페이지사상표를 참고해 페이지 프레임 번호 단위로 적재한다. v = (p, d) Ex) v(5, 3)이고 페이지 프레임 번호가 7이라면 r = (7 \* 5, 7 \* 5 + 3) , r은 실제 주소

**연관사상:** 연관 사상은 페이지 변환 정보를 연관 메모리에 저장된 연관사상표를 이용해 동적주소로 변환한다. 연관 메모리는 고속 액세스 메모리 장치이다. v(p,d)가 주어지면 연관 사상표에서 p를 찾아 p의 밸류에 해당하는 p’ + d로 실제 주소를 생성한다. 만약 연관사상표에 페이지 p에 대응하는 값이 없으면 직접 페이지 사상표를 탐색해 실주소를 생성한다. 이후 페이지 p가 p’에 위치하므로 연관 사상표가 이 정포를 포함하도록 갱신된다. 다음번에 동일한 프로세스가 호출될 때 페이지 p에 의한 참조는 연관사상표에 의해 빠르게 변환될 수 있다. 페이징 기법에서 프로세스 사이의 메모리 보호는 페이지 단위로 이뤄지며 메모리 크기도 페이지 프레임으로 나뉘어 외부 단편화가 발생하지 않는다. 내부 단편화는 발생할 수 있다.

**세그먼테이션 기법:** 다양한 크기의 세그먼트로 메모리를 나누어 관리하는 기법이다. v = (s, d).

위치지정에 최초 적합, 최적 적합 등의 방법이 사용된다. 페이지 사상 기법과 유사하게 세그먼트 사상표를 이용한다. 세그먼트 사상표에는 세그먼트 존재 비트, 보조기억장치 주소, 세그먼트 길이, 세그먼트 시작주소가 있다. 세그먼트 길이는 오버 플로를 확인할 때 사용한다. 만약 세그먼트 길이가 800인데 세그먼트 시작주소가 700이고 변위가 900이면 오버플로이다.

**페이징/세그먼테이션 혼용 기법:** 세그먼테이션 기법의 논리적 장점과 페이징 기법의 메모리 관리 측면의 장점을 활용하기 위한 기법이다. 각 세그먼트를 다시 페이지 단위로 분할하고 메모리도 페이지 프레임으로 분할해 하나의 페이지만 페이지 프레임에 적재한다. 가상주소 v에 대한 주소 지정은 v = (s, p, d).

**메모리 호출 기법:** 어느 시점에 페이지 또는 세그먼트를 메모리에 적재할 것인지 결정하는 기법이다. 페이징 기법의 호출 기법에는 요구 페이지 호출기법과 예상 페이지 호출기법이 있다.

**요구 페이지 호출기법: Demand Page Fetch Strategy** 프로세스의 페이지 요구가 있을 때 요구 페이지를 메모리에 적재한다. 프로세스 실행순서를 정확히 예측할 수 없기에 명령어나 데이터가 실제 참조되면 해당 페이지를 메모리에 적재한다. 오버헤드를 최소화하고 메모리에 옮겨진 페이지는 모두 페이지에 의해 실제 참조된 것임을 확신할 수 있다.

**예상 페이지 호출기법: Prepaging Fetch Strategy** 현 시점에 액세스되고 있지는 않지만 사용이 예상되는 페이지를 미리 메모리에 옮긴다. 예상이 맞으면 프로세스 단절이 없어 효율이 올라가고 틀리면 불필요한 시간과 메모리가 낭비된다. 시작 프로그램은 순수 요구 페이지 기법으로 하면 속도가 너무 느리고 연속 페이지 부재(Page Fault)가 발생하기 때문에 예상 페이지 호출기법을 많이 사용한다.

**10장. 페이지 교체 알고리즘**

**최적화 원칙(Principle of Optimality):** 운영체제는 최적화 원칙을 따라 페이지 프레임에 있는 페이지 중 가장 오랫동안 사용되지 않을 것 같은 페이지를 프로세스가 필요로 하는 페이지와 교체해 보조 기억장치에 보관한다.

**효율적 동작을 위해 페이지 교체를 하지 않는 영역:** 커널 코드 영역, 커널에 속하지 못한 보조기억장치 드라이버 영역, 시간에 맞춰 동작해야 하는 코드 영역, DMA(Direct Memory Access)등에 의해 입출력장치로부터 직접 데이터 교환이 되어야 하는 버퍼 영역

**FIFO 페이지 교체:** 큐를 사용해 메모리에서 가장 오래된 페이지를 교체한다. 실제 메모리에 가장 오래 머무른 페이지는 자주 사용되는 페이지일 확률이 높으므로 First In First Out 방식은 자주 사용하는 페이지를 교체할 위험이 있다. FIFO의 또 다른 단점으로는 Belady의 anomaly가 있다. 이는 프로세스에 더 많은 수의 페이지를 할당할 경우 오히려 페이지 부재가 더 많이 발생할 수 있다는 현상이다.

**LRU 페이지 교체:** Least Recently Used 교체 방법. 메모리에서 가장 오랫동안 사용되지 않는 페이지를 교체한다. 이는 국부성 휴리스틱에 의존하는 방식이다. 국부성은 시간 국부성과 공간 국부성으로 나뉜다. 시간 국부성은 시간적으로 가까운 시기에 참조된 메모리가 다시 참조될 확률이 높은 것이고 공간 국부성은 근래에 참조된 메모리와 가까이 있는 메모리가 계속 참조되는 경향을 말한다. LRU 알고리즘에는 두 가지 구현 방식이 있다.

첫 번째는 참조시간을 이용하는 방법이다. 각 페이지가 참조될 때마다 그때의 시간을 테이블에 기록한다. 페이지 교체 시기에 참조된 시간이 가장 이른, 가장 참조된 지 오래된 페이지와 새 페이지를 교체한다.

두 번째는 리스트를 이용하는 방법이다. 페이지가 액세스되면 해당 페이지를 리스트의 선두에 위치시킨다. 이렇게 되면 빈번하게 호출된 페이지일수록 선두에 위치하고 호출빈도가 낮은 프로세스는 말미에 위치한다. 페이지 교체 시 리스트의 끝에 있는 페이지가 교체 대상으로 선택된다.

LRU기법은 Belady의 이상현상이 발생하지 않는다. 많은 경우 최적화 원칙에 근사한 결과를 선택할 수 있는 방식이기도 하다. 하지만 휴리스틱에 의존함으로 틀릴 가능성도 내포한다. 하나의 프로세스가 여러 페이지로 구성되는 긴 루프를 가지고 있을 때 가장 오래 전에 호출된 페이지가 루프가 회전한 뒤 재호출될 수 있다. LRU는 이 루프의 첫 번째에 위치한 페이지들을 교체해버렸기 때문에 루프의 첫 페이지들을 전부 재호출해야 하는 경우가 생길 수 있다. 또한 LRU는 막대한 오버헤드를 초래한다. 매번 시간을 기록하고 참조시간을 조회, 그러한 리스트를 유지하는 것은 모두 구현상 오버헤드가 크다. 따라서 LRU 기법 자체는 별로 이용되지 않는다.

**LFU 페이지 교체:** Least Frequently Used 교체 방법. 빈도수에 의존하는 알고리즘이다. 참조 횟수를 카운팅해 페이지 교체 대상을 선정하기 때문에 가장 최근에 교체된 페이지일수록 교체될 확률이 높다. 최근에 호출되어 빈번하게 쓰이게 될 페이지인 경우 계속해서 페이지 교체와 재호출이 발생하게 된다. 또한 오래 전에 빈번하게 호출된 뒤 더이상 쓰일 일이 없는 페이지가 있더라도 해당 페이지가 메모리를 얼마간 지속적으로 점유하고 있게 돼 낭비가 초래된다. LRU 알고리즘과 마찬가지로 오버헤드도 크다. 참조 횟수를 카운팅하고 조회하는 모든 알고리즘은 구현상 오버헤드가 크다고 알아두자.

**2차 기회 페이지 교체:** Second Chance 교체 알고리즘. 참조 비트가 0이면서 메모리에서 가장 오래 있었던 페이지를 교체 대상으로 한다. FIFO에 참조 비트를 더해 2차 기회를 주는 알고리즘이다.

참조 비트란 페이지 프레임에 있는 페이지마다 부여된 값이다. 0으로 초기화되지만 참조될 경우 1이 된다. 모든 페이지의 참조 비트가 1이면 페이지 교체가 불가능하므로 0을 만드는 상황도 존재한다. 페이지 교체 시에 큐의 선두에서 페이지를 꺼내는 것은 FIFO와 같다. 하지만 해당 페이지의 참조 비트를 확인해 1일 경우 이를 0으로 초기화 하고 큐의 끝으로 보낸다. 0일 경우 해당 페이지와 새 페이지를 교체한다. 큐를 변형된 원형 큐로 관리하는 경우 특별히 클럭 페이지 교체 알고리즘이라고 한다. 원형 큐가 시곗바늘 돌듯 관리되기 때문이다.

원형 큐에서는 포인터의 위치와 참조 비트값을 바꿔주면 된다.

**프로세스별 페이지 집합관리**

프로세스 페이지 프레임의 개수는 시스템 성능에 영향을 미친다. 프로세스마다 사용 가능한 페이지 집합 크기가 다르다. 만약 페이지 집합이 작다면 메모리에 더 많은 프로세스를 적재할 수 있어 시스템 처리량을 증대 시킬 수 있다. 그러나 프로세스별 페이지 부재가 자주 발생해 성능이 저하될 수 있다. 반면 페이지 집합의 크기가 크면 메모리에 적재할 프로세스의 수는 줄어들지만 프로세스별 페이지 부재는 덜 발생한다. 이런 성질에 기반해 페이지 집합을 관리하는 알고리즘으로 워킹 세트 알고리즘과 PFF 알고리즘이 있다.

**워킹 세트 알고리즘:** working set란 한 프로세스가 최근 참조한 페이지의 집합이다. 데닝이 페이지 부재비율을 감소시키기 위해 제안한 모델이다. 프로세스의 워킹 세트 W(t, )는 시각 t에 t를 포함한 직전 시간 동안 참조한 페이지의 집합을 말한다. 는 워킹 세트의 윈도 크기이며, W(t, )는 결국 시각 t – + 1부터 시각 t까지 참조한 페이지의 집합이다.

집합이므로 윈도 크기가 고정되어도 워킹 세트의 크기가 달라질 수 있다. 하나의 프로세스가 효율적으로 수행되기 위해 해당 프로세스의 워킹 세트가 메모리에 계속 유지되어야 하며 그렇지 않은 경우 프로세스의 보조장치로부터 계속 페이지를 요구해 쓰레싱이 유발된다. 쓰레싱은 페이지 부재가 비정상적으로 많이 발생해 프로세스보다 페이지 교체에 너무 많은 시간을 소비해 시스템 처리량이 급격히 저하되는 것을 말한다.

워킹 세트 알고리즘은 프로세스의 워킹 트리를 메모리에 유지시키는 것을 원칙으로 한다. 프로세스마다 워킹 세트에 맞게 페이지 프레임 개수를 조절한다. 운영체제는 각 프로세스의 워킹 세트를 감시해 워킹 세트 크기에 해당하는 충분한 페이지 프레임을 할당한다. 만일 충분한 여분의 페이지 프레임이 존재하면 다른 프로세스를 들여와 실행 프로세스의 크기를 늘린다. 실행 중인 프로세스들의 워킹 세트 크기의 합이 증가해 총페이지 프레임 수를 넘어가면 운영체제는 우선순위가 가장 낮은 프로세스를 일시 중지시켜 여유 페이지 프레임을 확보한다. 워킹 세트에 포함되지 않는 페이지를 담고 있는 프레임은 필요시 교체 대상으로 선택할 수 있다.

워킹 세트는 과거를 통해 미래를 예측하는 알고리즘이기 때문에 틀릴 수도 있다. 또 워킹 세트를 정확히 파악해 이를 계속 갱신해 나가는 게 현실적으로 힘들다. 워킹 세트를 구하기 위한 윈도 크기 의 최적값을 알기 어려우며, 이 또한 변화할 수 있다는 문제점이 있다.

**PFF 알고리즘:** Page Fault Frequency, 페이지 부재 빈도를 이용해 프로세스별 페이지 집합의 크기를 변화시키는 기법. 페이지 부재 빈도는 직전 페이지 부재 이후로 경과된 시간의 역수로 계산한다. 페이지 부재 빈도의 상한과 하한을 정하고 페이지 부재 빈도가 상한보다 높으면 페이지 프레임의 개수를 1 증가시키고 하한보다 낮으면 그 사이에 참조하지 않았던 페이지를 모두 제거하고 페이지 프레임 개수도 줄인다. 프로세스별 페이지 집합이 워킹세트 알고리즘 만큼 자주 바뀌지 않고 페이지 부재가 발생하고 그때의 페이지 부재 빈도가 상한이나 하한을 넘어가는 경우에만 바뀐다.

**11장. 장치관리**

입출력장치는 일반적으로 세 범주, 전용장치(Dedicated Devices), 공용장치(Shared Devices), 가상장치(Virtual Devices)로 구분된다. 이 범주화는 단순히 장치 간의 기능적 특성 뿐만 아니라 장치관리자에 의해 어떻게 관리되는가 또한 고려한 결과이다.

**전용장치:** 한 번에 하나의 프로세스에만 할당되는 장치 -> 테이프 드라이브, 프린터, 플로터

**공용장치:** 여러 프로세스에 할당될 수 있는 장치 -> 디스크 팩

**가상장치:** 첫 번째와 두 번째의 조합 -> 디스크처럼 공유 가능한 장치를 이용해 전용장치를 공용장치처럼 보이게 해 여러 프로세스에 할당한다. 플로터에 디스크를 이용한 스풀링을 적용해 공유 가능한 가상의 플로터로 변화시키는 게 그 예이다.

**장치구성**

**논리적 구성:** 장치와 장치제어기가 하드웨어에 속한다면 장치 드라이버가 논리적 영역에 속한다. 장치 제어기가 장치를 직접적으로 다루는 전자장치(장치 내부 or 외부에 독립)라면 운영체제에 속한 장치 드라이버는 응용 프로그램이 요청한 입출력 요청을 해당 장치에 맞도록 변환한다. 장치 드라이버는 장비마다 다르며 보통 제조사에서 제공한다.

**물리적 구성:** CPU와 메모리, 장치는 버스로 연결되어 있어서 CPU는 장치에 명령을 보낼 수 있다. CPU는 이를 통해 장치제어기와 통신한다. 장치제어기는 각자 상태를 표시하는 레지스터를 갖고있다. CPU는 이 값을 통해 장치의 상태를 확인하고 여기에 값을 써서 장치에 명령할 수 있다.

또 메모리 사상 입출력(memory-mapping I/O) 방식의 소통도 있다. 메모리의 특정 영역을 장치제어기의 레지스터와 대응시키는 방식이다. CPU는 메모리에서 읽고 쓰는 일반적 명령을 통해 장치제어기와 직접 통신하는 것과 동일한 효과를 얻을 수 있다.

**입출력 처리 유형 – 입출력 발생시 세 가지 처리방법**

**프로그램 방법:** CPU만 이용하는 입출력 방법. Polling을 이용한다. 폴링은 CPU가 입출력상태를 계속 확인해 CPU가 원하는 상태가 될 때까지 기다리는 것이다. 될 때까지 재확인 하므로 CPU 낭비가 심하다.

**인터럽트 방법:** 입출력 처리에 인터럽트를 넣는 방법. CPU는 확인할 필요 없이 인터럽트 발생시 신호를 받고 작업한다.

입출력 장치가 가용 상태가 되면 입출력 장치는 인터럽트 제어기에 상태 신호를 보낸다. 인터럽트 제어기는 다시 CPU에 신호를 보낸다. 그렇게 되면 CPU는 현재 실행 중인 명령만 마치고 즉시 인터럽트에 응답하게 되고 이에 인터럽트 제어기는 이벤트 대상에 대한 정보를 CPU에게 보낸다. CPU는 현재 상태를 보관하고 필요한 입출력 처리를 끝낸 후 원래 프로세스 실행 상태로 복귀한다.

**DMA 방법:** Direct Memory Access는 DMA제어기를 통해 CPU를 통하지 않고 메모리에 직접 접근해 데이터를 전송한다. CPU는 입출력에 필요한 정보 -> 소스의 위치, 양, 목적지에 대한 정보 등을 DMA 제어기에 넘긴다. 소스와 목적지 중 하나는 메모리가 된다. DMA 제어기는 소스에서 목적지로 데이터를 보내도록 장치제어기에 요청하고, 이 과정을 CPU가 처음 지시한 양이 될 때까지 반복한다. 원하는 양의 입출력이 끝나면 DMA 제어기는 인터럽트 제어기에 신호를 보내 인터럽트를 발생시켜 CPU에 입출력 작업이 모두 끝났음을 알린다.

DMA 방식은 한 번에 입출력 량이 많을 경우 인터럽트 발생 횟수를 단 한 번으로 줄여 CPU의 효율을 증대시킨다. CPU와 DMA 제어기 모두 메모리를 액세스하기 때문에 동시에 엑세스 시도를 하면 충돌한다. 이때 DMA 제어기에 우선권을 주는데 CPU로부터 메모리 사이클을 훔쳐 낸다는 뜻에서 사이클 스틸링이라고 한다. 사이클 스틸링을 통해 입출력 장치의 효율이 높아지게 되는 것이다. 이런 논리로 입출력 위주 프로그램은 CPU 위주 프로그램보다 우선권을 받게 된다.

**입출력 관리**

**버퍼링:** 버퍼는 메모리의 일부로 입출력 데이터 등을 전송할 때 일시적인 저장소로 사용된다. 단일 버퍼링은 데이터가 입력되는 동안 처리될 수 없는 단점이 있다. 이중 버퍼링은 데이터의 저장과 처리가 동시에 일어날 수 있다. 이처럼 버퍼를 늘려서 여러 개를 돌려 쓰는 방식은 순환 버퍼링이라고 한다.

**스풀링:** 입출력 프로세스와 저속 입출력장치 사이의 데이터 전송을 자기 디스크 같은 고속장치를 통하도록 하는 것으로 일종의 버퍼링이다. 프린터에 스풀링을 적용하면 프로세스는 디스크에 데이터를 쓰고 추후 프린트가 가용 상태가 되면 디스크에 저장된 데이터를 인쇄하는 방식이다. 스풀링을 통하면 프로세스는 장치가 가용할 때까지 대기할 필요가 없다. SPOOL(Simultaneous Peripheral Operation On Line)에 인쇄할 데이터를 저장하는 것으로 프린트 작업이 끝나기 때문이다. 스풀링은 이렇게 독립적으로 사용해야 하는 장치를 여러 프로세스가 동시에 사용할 수 있는 가상장치로 변화시킨다.