OS

1장 운영체제 소개(최종수정일 7/27)

**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.