FINAL REPORT

(IC-PBL 1, 2, 3)

COM2005 운영체제론

TEAM NAME : 파이팀

컴퓨터학부 2021050300 김재민
컴퓨터학부 2021049298 박준철
인공지능학과 2021056908 수림봉
인공지능학과 2022059334 유천일

- 목차 -

1. CPU 스케줄링

1.1 서론

1.1.1 CPU 스케줄링의 필요성

멀티코어 프로세서가 데스크탑 시장에 보편화 되면서 단순히 증가된 코어 수 만큼의 성능 향상뿐만 아니라 여러 개의 코어수를 좀 더 효율적으로 활용하여 성능을 극대화 할 수 있는 전략이 필요하게 되었다. 이에 따라 운영체제 계층에서는 단순하게 증가된 코어에 프로세스를 스케줄링 하기보다, 여러 개의 코어로 인한 프로세스 공유 문제, 코어 간 로드의 불균형, 캐시 메모리의 지역성의 원칙 등을 고려하여 스케줄링을 좀 더 효율적으로 할 수 있어야 한다.

스케줄링 알고리즘에 따라 멀티코어 환경에서 프로세스가 준비 상태로부터 첫 실행을 시작하는 반응시간과 실행을 끝마치기까지의 반환시간이 많은 차이를 보인다. 따라서 사용자는 컴퓨터의 사용 용도에 따라 멀티 코어 환경에서 적절한 스케줄러를 선택하여야 한다.

스케줄링의 목적을 한마디로 정의하면 "응답시간이나 처리량, 효율성을 증대시키기 위해 CPU가 다음에 실행할 프로세스를 선택하는 것"이라고 할 수 있다.

만약 사용자가 주로 사용하는 프로세스가 CPU에게 서비스 받는 시간이 짧고, I/O를 기다리는 시간이 빈번한 입출력 중심 작업이라면, 반응 시간이 작을 수록 유리할 것이지만, 반대로 CPU 중심 작업이라면 반응성이 유리한 스케줄러보다는 전체 작업이 완료되는 반환 시간이 중요한 요인이 될 것이다.

스케줄링이 얼마나 효율적인지 판단하는 문제는 프로세스들이 일생동안 각종 대기 큐에서 대기 하는 시간을 얼마나 줄일 수 있을 것이냐 하는 문제와 일맥상통한다. 또한 근본적으로 대기 큐의 구조를 최적화 하는 문제도 스케줄링 성능에 중요한 요소이다.

1.2. CPU 스케줄링이란?

1.2.1 CPU 스케줄링의 정의

운영체제는 CPU를 프로세스 간에 교환함으로써, 컴퓨터를 보다 생산적으로 만든다. 다중 프로그래밍에서는 하나의 프로세스가 전형적으로 어떤 I/O 요청이 완료되기를 기다려야만 할 때까지 실행된다. 이렇게 되면 단순한 컴퓨터 시스템에서 CPU는 그저 놀고 있게 된다. 이러한 모든 대기 시간은 낭비되며, 어떠한 유용한 작업도 수행하지 못한다. 이러한 시간을 생산적으로 활용하기 위해, 운영체제는 다수의 프로세스를 메모리에 저장하고, CPU를 회수하여 여러 프로세스에 번갈아가며 할당하여 CPU가 쉬지 않게끔 하게 한다. 이 패턴을 설계하는 과정이 CPU 스케줄링이다.

1.2.2 CPU 스케줄링 기법

각 스케줄링 기법은 다음과 같은 사항을 고려하여 스케줄링 선택을 고려한다.

- CPU 이용률(Utilization) : CPU 이용률이 높을수록 좋다.
- 처리량(Throughput) : 처리 가능한 프로세스수가 많을 수록 좋다.
- 총처리 시간(Turnaround time) : 총 처리 시간이 작을 수록 좋다.
- 대기 시간(Waiting time) : 프로세스가 대기하는 시간이 작을 수록 좋다.
- 응답 시간(Response time) : 프로세스의 응답 시간이 작을 수록 좋다.

CPU 스케줄링은 대기중인 프로세스들 중 어느 프로세스에 CPU를 할당할 것인지를 결정하는 문제를 다룬다. 이 방법에 대해 대표적인 알고리즘이 존재한다.

1) FCFS(Fisrt come first served)
   - 시스템에 들어와 가장 먼저 들어온 프로세스를 선택한다.
2) SJF(Shortest job first)
   - 실행시간이 가장 짧은 프로세스가 먼저 CPU를 할당받는다.
3) RR(Round robin)
   - FCFS와 유사하지만, 시스템을 time slice라고 하는 작은 단위의 시간 내 실행이 가능하며, 그 시간 범위내에 시스템이 프로세스들 사이를 옮겨 다닐 수 있도록 선점이 추가되는 방식이다.
4) Priority Scheduling
   - 프로세스들에 우선순위를 부여하여 우선순위대로 CPU를 선점하는 방식

1.3. 리눅스 스케줄링

1.3.1 리눅스 스케줄링이란?

리눅스 스케줄링이란, 말그대로 리눅스 커널 내에서의 프로세스의 순서를 스케줄링하는 방법이다. 이때 위에서 설명한대로 CPU 스케줄링의 정의를 실천하지만, 리눅스 내에서의 스케줄링 기법은 상대적으로 공정성(fairness)를 많이 고려한 스케줄링 기법을 사용한다는 점에서 의의가 있다.

1.3.2 리눅스 스케줄링의 특징

리눅스의 스케줄러는 다음과 같은 특징을 가진다.

- 선점형 스케줄링(Preemptive scheduling)을 사용한다. 이는 한 프로세스가 CPU를 사용 중이라 하더라도, 더 높은 우선순위의 프로세스가 도착하면 현재 실행 중인 프로세스를 일시 중단하고 CPU를 새로운 프로세스에 할당하는 방식이다. 이 방식은 실시간 반응성과 사용자 인터랙션이 중요한 데스크탑 환경에서 매우 유리하다.

- 리눅스는 다중 스케줄링 정책을 지원한다. 대표적으로 일반 사용자 프로세스에는 Completely Fair Scheduler(CFS)를, 실시간 프로세스에는 SCHED_FIFO, SCHED_RR등의 정책을 적용할 수 있다.

- 리눅스 스케줄러는 멀티코어 및 NUMA(Non-Uniform Memory Access)환경에 최적화되어 있다. CPU 간의 작업 부하를 균등하게 분산시키는 load balancing 기능이 포함되어 있어, 특정 코어에 작업이 집중되는 현상을 방지하고 시스템 자원의 활용도를 극대화한다.

1.3.3 리눅스 스케줄링의 주요 요소

리눅스 스케줄러가 프로세스를 선택하고 실행 순서를 결정할 때 고려되는 핵심 요소가 있다.

- nice 값: nice는 일반 프로세스의 우선순위를 간접적으로 조절할 수 있는 사용자 접근 가능한 수치로, -20에서 +19까지의 값을 가진다. 값이 낮을수록 프로세스의 우선순위는 높아져 더 많은 CPU 시간을 할당받을 가능성이 높다. 이는 공정성을 유지하면서도 사용자 또는 시스템 관리자가 특정 작업의 중요도를 반영할 수 있도록 하는 기법이다.

- 우선순위(priority): 리눅스는 실시간(real-time)과 일반(nice-based) 프로세스를 분리하여 관리하며, 실시간 프로세스는 0~99 사이의 우선순위를, 일반 프로세스는 100~139의 우선순위를 가진다. 스케줄러는 이러한 우선순위를 기반으로 실행 대상을 선택한다.

- 시분할(time slicing): 모든 프로세스는 일정 시간 동안만 CPU를 사용할 수 있으며, 이를 time slice라 한다. time slice가 종료되면 스케줄러는 다른 프로세스에게 CPU를 양보하도록 강제하여 자원이 공평하게 분배되도록 한다.

- context switching: CPU가 하나의 프로세스에서 다른 프로세스로 전환되는 것을 컨텍스트 스위칭이라 한다. 이 과정에는 저장 및 복원 작업이 수반되므로 스케줄러는 최소한의 컨텍스트 스위치로 최대의 효율을 도모해야 한다.

1.3.4 O(1) 스케줄링 기법

리눅스 커널 2.5 버전에서는 O(1) 스케줄러가 도입되었다. 이 스케줄러는 이름 그대로 프로세스 수에 관계없이 항상 일정한 시간(O(1)) 내에 스케줄링 결정을 내릴 수 있도록 설계되었다.

O(1) 스케줄러는 프로세스를 우선순위 기반의 큐에 분리하여 저장하고, 가장 높은 우선순위의 큐에서 준비 상태의 프로세스를 선택하여 실행한다. 실행이 완료되거나 time slice를 모두 사용하면 해당 프로세스는 비활성 큐로 이동되고, 다음 프로세스가 선택된다. 이 과정에서 활성 큐와 비활성 큐를 교대로 교환하면서 모든 프로세스에게 실행 기회를 제공한다.

그러나 O(1) 스케줄러는 3가지 문제점을 드러냈다. 첫째, 낮은 우선순위의 프로세스가 장시간 실행되지 않는 기아(starvation)현상이 발생할 수 있었다. 둘째, 인터랙티브 프로세스의 응답성이 떨어져 사용자 경험이 저하되었다. 셋째, time slice가 우선순위에 따라 고정되어 있어, 워크로드 변화에 적응하기 어려운 구조적 한계가 있었다. 이로 인해 시스템이 비효율적으로 자원을 운용하는 경우가 늘어났고, 멀티코어 환경에서도 스케줄러의 유연성이 부족하다는 평가를 받았다.

1.3.5 CFS 스케줄링 기법

Completely Fair Scheduler(CFS)는 리눅스 커널 2.6.23 버전부터 기본 스케줄러로 채택된 알고리즘으로, 기존의 O(1) 스케줄러가 갖는 고정된 우선순위 체계와 비공정성 문제를 해소하기 위해 설계되었다. CFS는 프로세스 간의 공정한 CPU 시간 분배(fairness)를 핵심 목표로 삼으며, 이를 위해 고정된 우선순위 기반 대신 가상 런타임(virtual runtime, vruntime)이라는 상대적인 실행 지표를 기준으로 프로세스의 실행 순서를 결정한다.

CFS는 이상적인 상황에서 모든 프로세스가 동등한 실행 기회를 가져야 한다는 철학에 기반한다. CPU를 많이 사용한 프로세스는 당분간 뒤로 밀리고, 적게 사용한 프로세스에게는 우선적으로 CPU를 제공함으로써, 전체 시스템에서 공정한 CPU 분배를 실현하려는 구조이다. 이는 전통적인 우선순위 기반 방식과 달리, 실제 자원 사용량을 반영한 실행 순서를 산정하는 데 초점을 둔다.

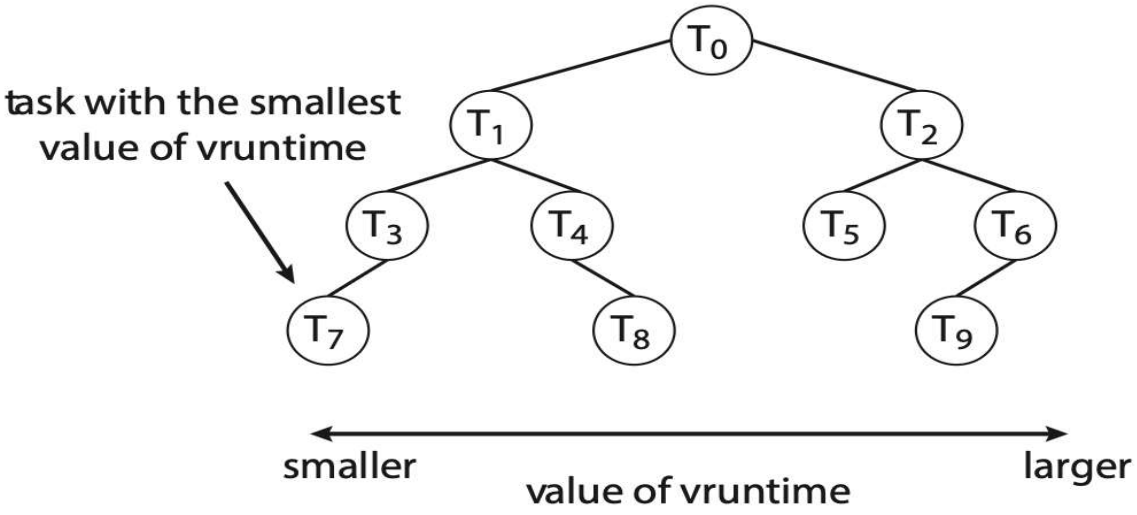
1.3.6 CFS 스케줄링 특징

- Red-Black Tree

CFS는 모든 실행 가능한 프로세스를 Red-Black Tree라는 균형 이진 탐색 트리에 저장한다. 이 트리에서 각 노드는 하나의 태스크(Task, 프로세스 또는 스레드)를 나타내며, 노드의 정렬 기준은 해당 태스크의 vruntime이다. vruntime이 가장 작은 노드(프로세스)가 루트에 가까이 위치하며, CPU는 이 태스크에 우선적으로 할당 된다. 또한 트리의 삽입, 삭제, 탐색에 대한 시간복잡도는 O(log n)을 갖는다.

이 구조는 CFS가 O(1) 스케줄링처럼 고정된 시간에 스케줄링 결정을 내릴 수

는 없지만, 공정성을 유지할 수 있다는 측면에서 큰 이점을 제공한다.



- vruntime과 nice값의 관계

vruntime은 실제 실행 시간을 기반으로 누적되는 값이지만, 단순한 누적이 아니라 nice값에 따라 가중(weighted)된 실행 시간이 적용된다. 다음과 같은 공식을 사용한다.

vruntime += 실행 시간 * (1024 / 가중치)

여기서 가중치는 프로세스의 nice값에 따라 정해진다. nice 값은 -20에서 +19까지 설정 가능하며, nice값이 낮을수록 높은 가중치를 가진다. (예: nice -5의 가중치는 약 1,400, nice 0의 가중치는 1024, nice 5의 가중치는 약 600) 따라서 nice값이 낮을수록 vruntime이 천천히 증가하므로, 해당 프로세스는 더 자주 CPU를 배정 받는다. 이는 우선순위가 높을수록 CPU를 더 많이 할당하는 효과를 낳지만, 기존처럼 정적인 우선순위가 아닌 실제 실행 시간 기반의 동적 우선조정을 가능하게 한다.

- CFS의 time slice 계산 방법

CFS는 타임 슬라이스(time slice)를 고정적으로 부여하지 않는다. 대신, 시스템 전체 부하(load)에 따라 각 태스크에 가변적으로 시간을 배분한다. 각 태스크가 차지하는 실행 시간 비율은 다음과 같이 계산한다.

프로세스 실행 시간 비율 = 프로세스 가중치 / 전체 태스크 가중치 합계

이를 통해 CFS는 시스템에 존재하는 모든 태스크가 상대적인 가중치에 비례하여 CPU 시간을 나눠 가지도록 설계되어 있다.

1.3.7 CFS 스케줄링 장/단점

1) 장점

- 공정성 확보

: 각 태스크의 자원 사용량에 따라 공정한 실행 순서가 결정된다.

- 유연한 우선순위 조절

: nice값을 통해 사용자가 직/간접적으로 우선순위를 조절 할 수 있다.

- 특정 프로세스 독점 방지

: CPU를 과도하게 사용하는 프로세스는 vruntime이 증가하여 실행이 늦춰진다.

- 전통적인 우선순위 문제 해결

: 고정 우선순위 체계에서 발생하던 기아문제 등의 문제를 효과적으로 예방할 수 있다.

2) 단점

- 정확한 실시간 응답이 요구되는 환경에서는 부적합

: CFS는 공정성을 기반으로 하기 때문에, 시간 제한을 보장하기 어렵다.

- 오버헤드 발생 가능성

: 많은 태스크가 동시에 실행 가능한 상태일 경우 오버헤드가 발생할 수 있다.

- 절대적인 우선순위 보장 불가

: 매우 낮은 nice값(높은 우선순위)이 있는 프로세스가 있더라도, 절대적으로 우선순위를 보장 할 수 없다.

3) 해결책

이러한 한계로 인해, 리눅스는 실시간 태스크(0~99의 범위)를 위한 SCHED_FIFO, SCHED_RR등의 별도 클래스도 병행적으로 지원한다.(100~139는 CFS사용)

1.4 결론

CPU 스케줄링은 운영체제의 핵심 기능 중 하나로, 한정된 CPU 자원을 효율적으로 분배하고 시스템의 전반적인 성능과 사용자 만족도를 좌우하는 중요한 요소이다. 특히 리눅스 운영체제는 다양한 하드웨어 환경과 사용자 요구에 대응하기 위해 유연하고 다층적인 스케줄링 구조를 발전시켜 왔다. 본 논문에서는 리눅스의 스케줄링 메커니즘을 중심으로, O(1) 스케줄러의 구조와 한계, 그리고 이를 대체한 Completely Fair Scheduler(CFS)의 원리를 중점적으로 살펴보았다.

O(1) 스케줄러는 고정된 우선순위 체계와 빠른 스케줄링 결정을 기반으로 우수한 시간 복잡도를 제공하였으나, 공정성과 확장성 측면에서 심각한 제약을 안고 있었다. 이에 반해 CFS는 공정성(fairness)을 중심으로 삼고, 프로세스별 가상 런타임(vruntime)을 기반으로 동적으로 실행 우선순위를 조정함으로써 기존의 문제를 효과적으로 개선하였다. Red-Black Tree 자료구조를 활용한 vruntime 기반 정렬, nice 값과 weight를 통한 실행 비율 조절 등은 CFS가 리눅스 커널에서 기본 스케줄러로 자리 잡는 데 중요한 기술적 기반이 되었다.

또한, 리눅스는 실시간 처리 요구에 대응하기 위해 SCHED_FIFO, SCHED_RR등의 스케줄링 클래스를 병행 운영함으로써, 일반 프로세스 환경과 실시간 프로세스 환경을 모두 포괄하는 유연한 구조를 구현하고 있다. 이러한 다중 클래스 기반 스케줄링은 다양한 우선순위 체계와 시간 제약 조건을 통합적으로 지원하며, 리눅스의 범용성과 확장성을 한층 강화하는 요소로 작용한다.

1.5 Q&A(멘토와의 질문)

Q. 첫 번째 파이팀의 질문 :

nice값이 같다는 전제하에, vruntime이 작은 태스크가 우선순위를 차지하는데, 그 역도 참이 될 수 있을까요? 즉 vruntime이 같고, nice값에 차이가 있는 경우가 있는지 궁금해요?

A. 멘토님의 답변 :

역을 이론적으로 참으로 볼 수는 있으나, 현재 그런 상황은 존재하지 않는다. nice 값이 같으면 vruntime이 작은 태스크가 우선순위를 가지는 것이 확실합니다. 반대로 vruntime이 같은 경우 nice값을 따지기보다는 그냥 자유롭게 처리하는 것이 현실적입니다.

Q. 두 번째 파이팀의 질문 :

nice값은 사용자가 임의로 조정 할 수 있을까요?

A. 멘토님의 답변 :

가능하다. 단 nice 값을 늘리는(우선순위를 낮추는) 것은 일반 사용자도 가능하지만, 줄이는(우선순위를 높이는) 것은 루트 권한이 필요하다.

Q. 세 번째 파이팀의 질문 :
1~99까지 실시간 프로세스의 우선순위를 부여하고, 100~139까지 일반 태스크의 우선순위를 부여할 때, 1~99까지의 범위에서 공간이 남는다면 100~139에 위치한 태스크를 당겨써도 되지 않을까요?

A. 멘토님의 답변 :
이론상 가능하다. 그러나 보통 그렇게 하지 않고 전체 우선순위 공간 크기를 줄이는 방식으로 활용한다. 물론 이 경우에도 루트권한이 필요하다.

2. 세마포어를 이용한 모니터 구현

2.1 동기화(synchronization)의 중요성

현대의 컴퓨터 시스템, 특히 멀티코어 및 멀티프로세싱 환경에서 여러 프로세스 혹은 스레드가 동시에 공유 자원에 접근하는 일이 빈번하다. 이때, 다중 스레드나 프로세스가 데이터의 동시에 접근하지 않기 위해 임계 구역(Critical section)을 설정해야 한다. 임계 구역 내에 다중 스레드나 프로세스가 동시에 접근하면 데이터의 상호 배제 문제(Mutual Exclusion), 실행되지 않는 데이터에 대한 기아 상태 문제(Starvation), 데드락(Deadlock), 우선순위 역전(Priority Inversion) 등 문제가 생기기 때문이다.

이러한 문제를 해결하기 위해 제공되는 이론이 동기화이다. 동기화는 데이터의 일관성과 시스템의 안정성을 보장하기 위한 필수 조건이다. 운영체제, 데이터베이스, 임베디드 시스템 등 다양한 분야에서 동기화는 항상 중요한 이슈로 다뤄지고 있다.

2.2 동기화 도구의 발전 배경과 모니터의 필요성

초기 동기화는 단순한 락(lock)이나 플래그(flag) 같은 방법으로 이루어졌다. 하지만 이런 저수준 동기화 방법은 프로그래머의 실수 위험이 높고 코드가 복잡해지는 단점이 있다. 이를 개선하기 위해 세마포어(semaphore), 뮤텍스(mutex), 이벤트(event), 그리고 한 단계 더 추상화된 모니터(monitor) 등 다양한 동기화 도구가 개발되었다. 특히 모니터는 동기화와 자원 관리의 복잡성을 추상화하여, 프로그래머가 더욱 안전하고 직관적으로 동기화 문제를 해결할 수 있도록 해준다.

우리는 이 프로젝트에서, 세마포어와 모니터를 적극적으로 활용해 동기화 문제를 해결할 수 있는 방법을 알아보고자 한다.

2.3 동기화 문제 사례 및 이용 방법과 고전적 해결

2.3.1 동기화 문제사례

1. Mars PathFinder 시스템 오류 (1997)

- NASA의 Mars PathFinder 탐사선이 자주 리부팅 되는 오류가 발생했다. 원인은 높은 우선순위인 스레드와 낮은 우선순위 스레드 사이에서 공유 자원에 대한 충돌이 있었기 때문이다. 우선순위 역전과 동기화에 대한 실패가 시스템 전체 오작동을 일으킨 대표적인 사례이다.

2. Therac-25 방사선 치료기 사고 (1985~1987)

- 동기화 오류로 인해서 방사선 치료기 기계가 치명적인 과다 방사선을 방출하게 된 사건이다. 공유 자원에 접근 하는 과정에서 적절한 동기화 기능의 부재로 인해서 우선순위 역전이 발생 한 것이 원인이다. 이 사건은 최소 6명의 환자의 중상 및 사망을 보도한 이력이 있는 동기화 문제의 대표적인 사례이다.

3. 은행 계좌 이체 시스템 오류

- 동시에 여러 트랜잭션이 한 계좌에서 출금하는 과정을 거치는 원인으로 데이터 불일치 상황이 발생했다. 적절한 뮤텍스(mutex)작업이 없어서 계좌 잔액 출력에 대해 오류를 발생시키고 말았다. 이로 인해서 고객의 잔액이 잘못 계산되어 금융사고가 발생하는 일이 빈번했다.

2.3.2 동기화 이용 방법

1. 임계 구역(critical section) 식별
 - 여러 스레드가 동시에 접근하면 문제가 생기는 임계 구역을 찾는다.
 예) 전역 변수, 공유 버퍼, 파일 등

2. 동기화 도구 선택
 - 문제 상황에 맞는 동기화 도구(예 : 세마포어, 뮤텍스, 모니터 등)를 고른다.
  → 단순 배타적 접근 : 뮤텍스 / 이진 세마포어
  → 여러 개의 자원 제어 : 카운팅 세마포어
  → 조건 동기화 : 모니터, 조건 변수 등

3. 공유 자원 진입/탈출을 알리기 위한 코드에 동기화 함수 삽입

- 임계 구역 진입 전에 동기화 도구의 "잠금" (sem_wait()) 연산을, 임계 구역 퇴장 후에 "해제" (sem_post()) 연산을 추가한다.

4. 스레드/프로세스 생성 및 동작

- 각 스레드나 프로세스는 동기화가 걸려 있는 함수나 코드 영역을 호출해서 동작한다.

2.3.3 동기화 문제의 고전적 해결법

1) 뮤텍스(Mutex, Mutual Exclusion)

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}

release() {
    available = true;
}
```

- 임계 구역을 하나의 스레드만 접근하도록 "상호 배제"를 보장한다.
- available을 통해 임계 구역으로의 접근 허용 / 차단을 보장해준다.
- 대표적인 소프트웨어적 / 하드웨어적 방법:

→ Dekker's Algorithm (데커의 알고리즘)
→ Peterson's Algorithm (피터슨 알고리즘)
→ Test-and-Set, Swap, Compare-and-Swap 등 (하드웨어 명령)

2) 세마포어(Semaphore)
- 1965년 Dijkstra가 제안한 동기화 도구이다.
- 카운팅, 이진 세마포어 등 여러 동기화 상황에 사용이 가능하다.
- 임계 구역, 생산자-소비자 문제 등에서 많이 활용하고 있다.

3) 모니터(Monitor)
- 현재 제시된 가장 최적의 고급 동기화 기법 도구 중 하나이다.
- 임계 구역 코드와 동기화 메커니즘을 하나의 모듈로 감싸는 방식이다.
- 조건 변수(Condition Variable)와 같이 사용한다.

4) 스핀 락(Spin Lock)
- 짧은 임계 구역에 대해 락(Lock)이 풀릴 때까지 바쁘게 대기하는 기법이기에 바쁜 대기(busy-waiting) 기반으로 작동한다.

2.4. 세마포어(Semaphore)의 개념

2.4.1 세마포어(Semaphore)란?

- 세마포어란 동시성 제이를 하기 위한 변수로, 자원의 수를 나타내며, 공유 자원의 데이터 혹은 임계 구역에 대해서 다중 스레드가 접근 하는 것을 제어하는 역할을 한다.

2.4.2 세마포어(Semaphore)의 구조 및 특징

(1) 세마포어는 임계 구역 보호를 위해 도입된 동기화 도구로, 주로 두 가지 연산으로 동작한다.

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

- P(wait) 연산:
세마포어 값(value)을 1 감소시키며, 값이 음수가 되면 스레드는 대기(block)한다.

- V(signal) 연산:
세마포어 값(value)을 1 증가시키고, 대기 중인 스레드가 있으면 하나를 깨운다.(wakeup)

(2) 세마포어 사용 특징에 대해서는 사용 방법에 따라 Counting 방식, Binary 방식으로 나뉜다.

- Counting semaphore(계수형 세마포어):
0 이상의 정수 값을 가지며, 여러 개의 동일 자원(예: 프린터 여러 대 등)에 대한 접근을 제어할 때 사용된다. 세마포어 값이 자원의 개수를 나타내며, 여러 스레드가 동시에 자원을 사용할 수 있다.

- Binary semaphore(이진 세마포어):
값이 0 또는 1만을 가지는 세마포어로, 사실상 뮤텍스와 동일하게 동작한다. 오직 하나의 스레드만 임계 구역에 진입할 수 있도록 제한한다.

2.4.3 세마포어의 한계

- P/V 연산의 순서를 잘못 사용할 경우 데드락(DeadLock), 우선순위 역전(Priority Inversion), 기아(starvation) 현상 등이 쉽게 발생할 수 있다.

- 동기화에 대한 책임이 전적으로 프로그래머에게 있기 때문에 실수 위험이 높다.

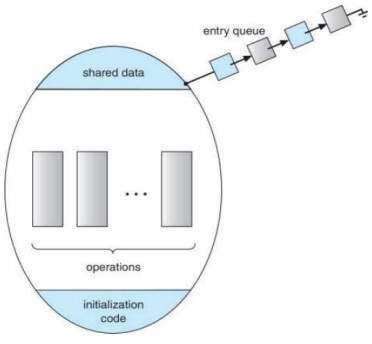
- 생산자-소비자 문제와 같은 복잡한 조건 동기화 상황에서는 코드가 복잡해지고, 실수로 인한 버그 발생 확률이 크게 증가한다.

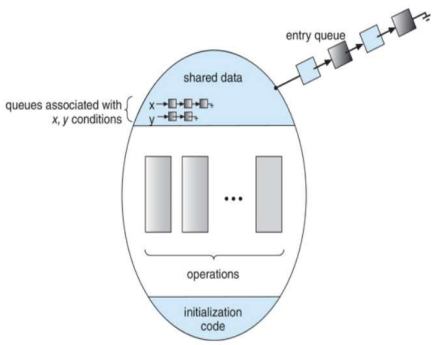
2.5. 모니터(Monitor)의 개념

2.5.1 모니터란?

세마포어를 사용하더라도, 연산 순서의 대한 오류가 존재 할 수 있다(예: wait(), signal()의 연산 순서 바뀌는 경우). 이러한 오류를 해결하기 위해 나타난 도구가 모니터이다. 모니터는 동기화와 데이터 보호를 캡슐화한 고수준 추상 데이터 타입이다. 모니터는 공유 자원과 그 자원에 대한 연산(메서드), 그리고 조건 변수(Condition Variable)를 하나의 모듈로 캡슐화하며, 모니터 내의 함수들은 상호배제를 자동으로 보장한다. 따라서 프로세스나 스레드는 외부에서 오직 모니터가 제공하는 공개 함수(메서드)를 통해서만 자원에 접근할 수 있다.

2.5.2 모니터의 구조 및 특징





- 진입 대기열(Entry Queue):
모니터에 진입을 요청하는 스레드들은 진입 대기열에서 대기하다가, 모니터가 비어 있을 시, 큐에 대기한 순서대로 진입한다.

- 조건 변수별 대기열(Condition Queue):
각 조건 변수(x, y 등)에 대해 별도의 대기열이 있어, 특정 조건이 충족될 때까지 스레드가 대기한다.

- 초기화 코드(Initialization Code):
모니터 내부에 있는 데이터와 변수는 초기화 코드에 의해 설정되어 모니터 생성 시 한 번만 실행된다.

- 연산(Operations)과 공유 데이터(Shared Data)의 일체화:
연산들은 반드시 모니터 내부의 공유 데이터에만 접근할 수 있다. 외부에서 직접 공유 데이터 접근은 불가하다.

- 모니터의 재진입 불가:
이미 어떤 스레드가 모니터 내에 있으면, 다른 스레드는 반드시 진입 대기열에서 기다려야 한다.

- 모듈화 및 구조화:
모니터는 데이터와 동작을 하나의 단위로 묶어, 프로그램의 구조적 설계에 도움을 준다.

- 직관적 동기화 흐름:
조건 변수의 wait/signal 메커니즘 덕분에 동기화 로직이 더 명확하고 이해하기 쉽다.

2.5.3 모니터와 세마포어의 차이

구분	세마포어	모니터
추상화 수준	낮음	높음
상호배제	수동(P/V)	자동(내장)
조건변수	없음	제공
프로그래머 오류	많음	적음
코드 가독성	낮음	높음

세마포어와 모니터는 모두 동시성 제어를 위한 대표적인 동기화 도구지만, 설계 방식과 프로그래밍 경험에서 큰 차이가 있다. 세마포어는 비교적 단순하고 저수준에서 유연하게 사용할 수 있지만, 그만큼 프로그래머가 직접 모든 동기화 절차를 신경 써야 하므로 실수나 버그의 위험이 크다.

반면, 모니터는 내부적으로 상호배제와 조건 동기화를 자동으로 처리해주기 때문에, 복잡한 동기화 상황에서도 코드가 간결하고, 오류 가능성도 줄어드는 장점이 있다. 실제로 운영체제, Java 등 여러 현대적 언어에서 모니터 기반의 동기화 메커니즘(예: synchronized, lock 등)을 기본적으로 지원하는 이유도 이러한 안정성과 추상화의 장점 때문이다.

결국, 단순한 동기화가 필요하고 성능이 중요한 경우에는 세마포어가, 코드의 안전성과 유지보수가 더 중요한 상황에서는 모니터가 더 적합하다고 할 수 있다.

2.6. 세마포어를 이용한 모니터의 구현

2.6.1 모니터 내부 구조의 세마포어 변환 원리

실제로 운영체제나 라이브러리에서 세마포어와 조건변수를 조합하여 모니터를 구현할 수 있다.

- 상호배제 : 뮤텍스로(이진 방식으로 사용) 구현
- 조건변수 : 세마포어 + 대기 큐로 구현

2.6.2 모니터의 기본적인 작동 방식

1) mutex 세마포어 초기화:
세마포어를 이용하여 모니터를 구현할 때, mutex 세마포어와 조건 변수를 핵심적으로 활용한다. 모니터에 처음 접근할 때 mutex 세마포어를 1로 초기화한다. 이는 critical section에 한 번에 하나의 스레드만 접근할 수 있도록 허용하는 역할을 한다. 즉, 모니터의 진입점 역할을 수행한다.

2) critical section 진입 및 작업 수행:
특정 스레드가 critical section에 진입하기 위해 mutex 세마포어에 대한 wait() 연산을 수행하면, mutex 값은 0으로 변경되어 다른 스레드들의 접근을 차단하고 해당 스레드가 작업을 수행할 수 있도록 대기 상태를 만든다.

3) 대기 중인 작업 처리 (next_count 활용):
critical section에서 작업을 마친 스레드가 signal() 연산을 수행할 때, next_count라는 변수를 확인한다. next_count는 모니터 내에서 작업이 완료되기

를 기다리는 스레드(주로 조건 변수에서 대기 중인 스레드)의 수를 나타낸다.
- next_count가 0보다 크다면, 즉 대기 중인 작업이 있다면 next 세마포어에 대한 signal() 연산을 수행하여 다음 대기 중인 스레드가 작업을 수행하도록 허용한다. 이는 모니터 내에서 특정 조건이 충족되기를 기다리는 스레드들에게 우선권을 부여하는 방식이다.
- next_count가 0이라면, 즉 더 이상 대기 중인 작업이 없다면 mutex 세마포어에 대한 signal() 연산을 수행하여 다른 스레드들이 다시 임계 구역에 접근할 수 있도록 허용한다.

2.6.3 모니터 변수의 활용

모니터 내에서 스레드들은 특정 조건이 충족될 때까지 대기할 수 있다. 이때 사용되는 것이 조건 변수이다. 조건 변수는 x_sem(조건 변수 x와 연관된 세마포어)과 x_count(조건 변수 x에서 대기 중인 스레드 수)로 구성된다.

1) 조건 변수 대기:
현재 스레드가 특정 조건 변수 x에서 대기해야 할 경우, 해당 스레드는 조건 변수에 대한 wait() 연산을 수행한다.

2) 대기 스레드 수 증가 및 critical section 진입:
wait() 연산 시 x_count를 증가시켜 조건 변수에서 대기 중인 스레드가 있음을 표시한다. 이어서 mutex를 통해 critical section에 진입하여 작업을 수행한다.

3) 작업 완료 후 처리:
critical section 내에서 작업을 마친 스레드는 남아있는 작업의 여부와 조건 변수 대기 상태를 확인한다.
- 남아있는 작업이 있으면: 다음 작업을 수행하도록 signal() 연산을 수행한다.
- 남아있는 작업이 없으면: mutex를 통해 다른 스레드에게 critical section 접근을 허용하고, 자신은 x_sem에서 대기한다.

4) 대기 스레드 수 감소:
만약 작업이 모두 끝나 더 이상 조건 변수에서 대기할 필요가 없는 상태가 되면, x_count를 감소시켜 대기 중인 스레드 수를 갱신한다.

2.6.4 세마포어 기반 모니터 설계 예제

```
typedef struct {
    sem_t mutex;            // 모니터 상호배제용 세마포어
    sem_t cond;             // 조건 변수용 세마포어
    int cond_waiting;       // 조건 대기 중 스레드 수
    int data;               // 공유 데이터 예시
} monitor_t;

void monitor_init(monitor_t *m) {
    sem_init(&m->mutex, 0, 1);
    sem_init(&m->cond, 0, 0);
    m->cond_waiting =0;
}
```

이 구조는 세마포어를 이용해 모니터의 기본 기능(상호배제, 조건 동기화)을 직접 구현하려는 패턴이다. 실제로 이 구조를 활용하려면 아래와 같이 적절히 조합하여 wait, signal 연산을 별도의 함수로 구현해야 한다.

- 진입/퇴장 시 sem_wait(&m->mutex)와 sem_post(&m->mutex)
- 조건 대기 및 신호 시 cond 세마포어와 cond_waiting 변수

실제 C 언어나 POSIX 환경에서는 모니터가 언어 차원에서 지원되지 않기 때문에, 이런 식으로 세마포어를 이용해 모니터와 유사한 동기화 패턴을 구현할 수 있다.

2.6.5 조건 변수와 세마포어의 결합

조건변수는 대기 중인 스레드를 세마포어 큐에 넣고, signal에서 꺼내 복귀시킨다.

1) monitor_wait(monitor_t *m)

```
void monitor_wait(monitor_t *m) {
    m->cond_waiting++;
    sem_post(&m->mutex);          // 임계구역 탈출
    sem_wait(&m->cond);           // 조건 변수에서 대기
    m->cond_waiting--;
    sem_wait(&m->mutex);          // 재진입
}
```

대기자 수 증가 : m->cond_waiting++

- 조건 변수를 기다리는 스레드 수를 1 증가시킨다.

임계구역 탈출 : sem_post(&m->mutex)
- 현재 임계 구역(모니터)을 다른 스레드가 사용할 수 있도록 잠금을 해제한다.
- 조건 변수를 기다리는 동안에는 임계 구역에 머무르면 안 되기 때문이다.

조건 변수에서 대기 : sem_wait(&m->cond)
- 조건 변수가 충족될 때까지(즉, signal이 호출될 때까지) 현재 스레드는 대기 상태로 전환된다.

대기자 수 감소 : m->cond_waiting--
- 조건이 충족되어 깨어난 스레드는, 대기 중인 스레드 수를 감소시킨다.

임계구역 재진입 : sem_wait(&m->mutex)
- 다시 임계 구역(모니터)에 진입하여, 이어서 작업을 수행합니다.

2) monitor_signal(monitor_t *m)

```
void monitor_signal(monitor_t *m) {
    if (m->cond_waiting >0)
        sem_post(&m->cond);      // 대기자 호출
}
```

대기자 호출:
- 조건 변수를 기다리는 스레드가 하나라도 있다면(m->cond_waiting > 0), sem_post(&m->cond)를 통해 대기 큐에 있는 스레드 하나를 깨운다.
- 깨어난 스레드는 monitor_wait의 다음 단계로 진행하여 임계 구역에 재진입이 가능하다.

2.6.6 실제 코드(생산자-소비자 문제)

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define BUF_SIZE 5

typedef struct {
    int buf[BUF_SIZE];
    int count, in, out;
    sem_t mutex;            // 모니터 상호배제
    sem_t not_full;         // 조건 변수: 버퍼가 꽉 차지 않음
    sem_t not_empty;        // 조건 변수: 버퍼가 비지 않음
} monitor_t;

monitor_t monitor;

void monitor_init(monitor_t *m) {
    sem_init(&m->mutex, 0, 1);
    sem_init(&m->not_full, 0, BUF_SIZE);
    sem_init(&m->not_empty, 0, 0);
    m->count = m->in = m->out =0;
}

void produce(int item) {
    sem_wait(&monitor.not_full); // not_full 조건 대기
    sem_wait(&monitor.mutex);    // 모니터 진입
    // 버퍼에 아이템 추가
    monitor.buf[monitor.in] = item;
    monitor.in = (monitor.in +1) % BUF_SIZE;
    monitor.count++;
    printf("생산: %d (버퍼: %d개)₩n", item, monitor.count);
    sem_post(&monitor.mutex);    // 모니터 탈출
    sem_post(&monitor.not_empty); // not_empty 조건 신호
}

int consume() {
    int item;
    sem_wait(&monitor.not_empty); // not_empty 조건 대기
    sem_wait(&monitor.mutex);     // 모니터 진입
    // 버퍼에서 아이템 꺼냄
    item = monitor.buf[monitor.out];
    monitor.out = (monitor.out +1) % BUF_SIZE;
    monitor.count--;
    printf("소비: %d (버퍼: %d개)₩n", item, monitor.count);
    sem_post(&monitor.mutex);    // 모니터 탈출
    sem_post(&monitor.not_full); // not_full 조건 신호
    return item;
}
```

```
void* producer(void *arg) {
    for (int i =1; i <=10; i++) {
        produce(i);
        usleep(100000);
    }
    return NULL;
}

void* consumer(void *arg) {
    for (int i =1; i <=10; i++) {
        consume();
        usleep(200000);
    }
    return NULL;
}

int main() {
    pthread_t p, c;
    monitor_init(&monitor);
    pthread_create(&p, NULL, producer, NULL);
    pthread_create(&c, NULL, consumer, NULL);
    pthread_join(p, NULL);
    pthread_join(c, NULL);
    return 0;
}
```

<생산자 코드 (produce 함수)의 동작>

1) 버퍼 공간 확인

sem_wait(&monitor.not_full);

→ 만약 버퍼가 가득 차 있으면, 여기서 생산자는 대기(블로킹)한다.

→ 공간이 있으면 다음 단계로 진행.

2) 상호배제 진입

sem_wait(&monitor.mutex);

→ 여러 스레드가 동시에 버퍼에 접근하지 못하게 락을 건다.

3) 아이템 추가

버퍼에 아이템을 넣고, 인덱스/카운트 값을 갱신한다.

4) 상호배제 해제

sem_post(&monitor.mutex);

→ 버퍼 사용이 끝났으니 락을 푼다.

5) 소비자 신호(not_empty)
sem_post(&monitor.not_empty);
→ 소비자가 대기 중이었다면, 버퍼에 아이템이 생겼으니 소비자를 깨운다.

<소비자 코드 (consume 함수)의 동작>
1) 버퍼 비었는지 확인
sem_wait(&monitor.not_empty);
→ 버퍼에 아이템이 없으면, 소비자는 여기서 대기(블로킹)한다.
→ 아이템이 있으면 다음 단계로 진행.

2) 상호배제 진입
sem_wait(&monitor.mutex);
→ 여러 스레드가 동시에 버퍼에 접근하지 못하게 락을 건다.

3) 아이템 소비
버퍼에서 아이템을 꺼내고, 인덱스/카운트 값을 갱신한다.

4) 상호배제 해제
sem_post(&monitor.mutex);
→ 버퍼 사용이 끝났으니 락을 푼다.

5) 생산자 신호(not_full)
sem_post(&monitor.not_full);
→ 생산자가 대기 중이었다면, 버퍼에 공간이 생겼으니 생산자를 깨운다.

<생산자-소비자 문제 코드 요약>
- 생산자는 버퍼에 공간이 없으면 기다리고, 아이템을 넣은 뒤 소비자에게 신호를 보낸다.
- 소비자는 버퍼에 아이템이 없으면 기다리고, 아이템을 꺼낸 뒤 생산자에게 신호를 보낸다.
- 둘 다 버퍼 접근 전후에 mutex 세마포어로 상호배제를 보장해서 경쟁 조건을 막는다.

<생산자-소비자 문제의 전형적 해결>
- 이 코드는 대표적인 동기화 문제인 생산자-소비자(Producer-Consumer) 문제를 모니터 패턴과 세마포어로 구현한 예제이다.
- 생산자(producer)는 버퍼가 가득 차면 대기하고, 소비자(consumer)는 버퍼가

비면 대기하게 되어, 버퍼 오버플로우·언더플로우를 모두 방지한다.

<모니터 패턴에 충실한 구현>
- monitor_t 구조체는 내부적으로 데이터(버퍼 및 인덱스, 카운트)와 동기화 객체(세마포어)를 모두 은닉하고 있다.
- 버퍼 접근은 반드시 produce와 consume이라는 두 함수(=모니터 메서드)를 통해서만 이루어진다.
- 외부에서는 절대로 내부 상태를 직접 건드릴 수 없다.
- 캡슐화와 일관성 유지라는 모니터의 핵심 특징이 잘 구현되어 있다.

<세마포어를 활용한 조건 변수 구현>
- not_full과 not_empty 세마포어는 각각 버퍼가 가득 차지 않은 상태와 비어 있지 않은 상태라는 조건 변수를 대체한다.
- 버퍼에 공간이 없으면 생산자는 not_full에서 블록되고, 버퍼에 아이템이 없으면 소비자는 not_empty에서 막힌다(block).
- 각 조건 변수는 세마포어 카운트를 이용해 조건 만족 여부를 자동으로 관리한다.

<상호배제의 보장>
- mutex 세마포어는 버퍼(공유 데이터)에 대한 동시 접근을 차단한다.
- 반드시 한 번에 한 스레드만 버퍼에 접근할 수 있어, 경쟁 조건(race condition)이 발생하지 않는다.
<페어 신호 방식의 장점>
- 각 조건 변수 신호(not_full, not_empty)는 해당 조건을 만족할 때만 올려주기에 불필요한 깨우기(Wakeup)가 없다.
- 효율적이며 데드락 위험도 줄어든다.

<실행 흐름의 직관성>
- 생산자와 소비자는 각각 자기 역할에 맞는 조건 변수에서만 대기/신호를 수행한다.
- 실제 문제에서 필요한 동기화 로직이 함수로 명확하게 구분되어 있어, 코드의 가독성과 유지보수성이 높다.

<한계 및 확장 가능성>
- 현재 코드는 단일 생산자, 단일 소비자만 가정하고 있다. 다중 생산자/소비자가 동시에 동작해도 논리적으로는 문제 없지만, 스레드 수가 많아지면 성능이나 우선순위 역전 등의 고려가 추가적으로 필요할 수 있다.
- 버퍼 사이즈와 반복 횟수는 상수로 고정되어 있지만, 일반화도 쉽게 가능하다.

2.7 모니터(Monitor)의 구현의 설계 분석

2.7.1 세마포어만 사용할 때의 한계

세마포어는 저수준 동기화 도구로 유연성이 크지만, 다음과 같은 한계가 있다.

- wait/signal 실수로 인한 데드락 및 논리적 오류:
세마포어 연산(P/V)을 잘못 배치하거나 한 번이라도 누락하면, 전체 프로그램이 멈추거나 공유 데이터에 비일관성이 생길 수 있다.
(예시: 생산자-소비자 문제에서 signal을 빼먹으면 소비자가 영원히 깨어나지 못한다.)

- 신호(signal) 누락으로 인한 기아 상태 발생:
깨어나야 할 스레드에게 signal을 보내지 않으면, 일부 스레드가 무한 대기 상태에 빠져버릴 수 있다.

- 여러 조건이 복합적으로 얽힐 경우 코딩 난이도 급증:
예를 들어, "버퍼가 비었을 때"와 "버퍼가 가득 찼을 때"처럼 여러 조건이 얽혀 있으면, 세마포어만으로는 각 조건별로 정확히 wait/signal을 관리해야 되기에 코드가 복잡해지고, 실수 위험이 크다.

2.7.2 모니터의 장점

모니터는 고수준 동기화 추상화로, 아래와 같은 강점이 있다.

- 자동 상호배제:
모니터의 진입/퇴장 자체에 상호배제 로직이 내장되어 있어, 프로그래머가 일일이 lock/unlock을 신경 쓸 필요가 없다.

- 코드 가독성/유지보수 용이:
공유 데이터와 동기화 메서드가 하나의 구조체 또는 객체에 캡슐화되어 있어, 프로그램 흐름이 명확하고 코드를 읽기 쉽다.

- 조건 변수 지원:
복잡한 조건 동기화(예: 생산자-소비자, 독자-작가 문제)를 보다 명확하고 직관적으로 표현할 수 있습니다. wait/signal 연산이 조건 변수로 분리되어 논리적 오류가 줄어든다.

- 프로그램 신뢰성:
동기화 실수 가능성이 줄어들어, 데드락 등 치명적 오류를 예방할 수 있다.

2.7.3 세마포어를 기반으로 모니터를 구현할 때의 설계상 장점

언어 차원에서 모니터를 지원하지 않는 환경에서는 세마포어를 활용해 고수준 동기화 추상화를 흉내낼 수 있다.

- 기존 OS/라이브러리의 세마포어 자원 재활용:
별도의 커스텀 동기화 도구 없이 표준 세마포어만으로도 안전한 동기화 모듈을 설계할 수 있다.

- 고수준 모니터 추상화로 코드 일관성/안정성 확보:
프로그래머가 일관된 방식으로 공유 자원에 접근할 수 있어, 프로그램 구조가 명확해지고 유지보수가 쉬워진다.

- 언어 지원이 부족한 환경(C, C++ 등)에서도 구현 가능:
Java, Python 등에서는 모니터가 내장되어 있지만, C/C++에서는 직접 구현해야 하므로 세마포어를 조합해 모니터 패턴을 구현하는 것이 유용하다.

2.7.4 고려사항 및 한계(데드락, 우선순위 역전 등)

세마포어로 구현한 모니터도 아래와 같은 한계를 완전히 벗어날 수는 없다.

- 여전히 조건변수 wait/signal 실수 시 데드락 위험:
조건 변수에 대해 wait/signal 연산을 잘못 사용하면, 프로그램이 멈출 수 있다.
예시: signal을 빼먹거나, 잘못된 조건에서 wait을 호출하는 경우.

- 우선순위 역전(Priority Inversion):
낮은 우선순위의 스레드가 모니터(락)를 점유 중일 때, 높은 우선순위 스레드가 대기해야 하는 상황이 발생할 수 있다. 이를 해결하려면 우선순위 상속(priority inheritance) 등 추가 기법이 필요하다.

- 공정성 보장:
특정 스레드가 계속 깨어나거나(Starvation), 나머지 스레드가 무한 대기하는 문제가 발생할 수 있다. 따라서 공정성 보장을 위한 notifyAll(전체 깨우기) 전략이나, 대기열 순서 관리가 필요하다.

2.7.5 실제 운영체제 및 언어에서의 적용 사례

실제 시스템과 프로그래밍 언어에서는 다음과 같이 동기화 패턴이 활용된다.

- Java, C#, Python 등:
언어 차원에서 모니터와 조건 변수를 내장 지원한다. 예를 들어 Java의 synchronized 키워드, C#의 lock, Python의 threading.Condition 등이 대표적 예시이다.

- Unix/Linux 커널:
사용자 레벨에서는 뮤텍스, 세마포어, 스핀락, wait queue 등을 조합해 동기화를 진행한다. 커널 내부적으로도 모니터와 유사한 추상화로 자원 접근을 보호한다.

- 대표 동기화 사례:
생산자-소비자(Producer-Consumer), 독자-작가(Reader-Writer) 문제, 데이터베이스의 트랜잭션 동기화, 네트워크 소켓 버퍼 관리 등에서 널리 활용된다.

2.8 결론

2.8.1 정리 및 요약

모니터는 세마포어만을 사용할 때 발생하는 복잡한 코드 구조와 동기화 실수(데드락, 기아, 경합 등)의 위험을 효과적으로 해결하는 고수준 동기화 추상화 도구입니다. 모니터 내부에 공유 데이터와 동기화 연산(메서드), 조건 변수를 함께 캡슐화함으로써, 프로그래머가 일일이 동기화 연산을 관리할 필요 없이 코드의 안전성과 가독성을 크게 높일 수 있다.

특히, C/C++처럼 언어 차원의 모니터 지원이 없는 환경에서는 세마포어를 조합해 모니터 패턴을 직접 구현함으로써 기존 시스템의 세마포어 자원을 재활용하고 운영체제 및 응용 소프트웨어에서 요구하는 신뢰성과 효율성 모두를 만족시킬 수 있다.

이로써, 모니터 기반의 동기화 구조는 실제 시스템 개발과 유지보수 과정에서 오류 위험을 줄이고, 구조적 설계와 확장성 측면에서도 뛰어난 성과를 보여준다.

2.8.2 앞으로의 동기화 도구 발전 전망

미래의 컴퓨팅 환경은 멀티코어 CPU의 확산, 대규모 병렬처리, 분산 시스템, 클라우드 기반 컴퓨팅 등 점점 더 복잡한 동시성 환경으로 발전하고 있다.
이에 따라, 기존의 저수준 동기화 도구(세마포어, 락 등)만으로는 관리하기 어려운 다양한 동기화 요구가 발생하고 있다. 모니터와 같은 고수준 동기화 추상화는 앞으로도 중심적인 역할을 하게 될 것이다.

또한, 언어와 운영체제 차원에서 모니터 및 조건 변수, 트랜잭션 메모리 등 내장 동기화 기능이 계속해서 강화될 전망으로 보인다. 자동화 도구(예: 동기화 오류 검출, 데드락 분석, 경합 조건 자동 테스트 등)와 정형 검증 기법(formal verification)을 통한 프로그램의 동시성 안전성 보장 기술이 함께 발전할 것이다.

이러한 발전을 바탕으로, 미래에는 개발자가 동기화 실수로 인한 문제를 걱정하지 않고 더욱 안전하고, 효율적이며, 확장 가능한 소프트웨어를 설계할 수 있을 것으로 기대된다.

2.9 멘토님과의 튜터링

Q. 첫 번째 파이팀의 질문 :

세마포어를 이용한 모니터 구현 시, 기아 현상을 방지하기 위해 wait() 연산에 타임아웃을 설정하여 일정 시간 동안 자원을 획득하지 못한 스레드가 대기 큐에서 제거되고 재시도하거나 대체 경로를 선택하도록 하는 방식은 효율적일까요?

A. 멘토님의 답변 :

타임아웃 설정을 통한 기아 현상 방지 노력은 상황에 따라 효율성이 달라집니다.

1) 긍정적인 측면에서 볼 때, 타임아웃 설정은 다음과 같은 이점을 제공
- 부분적인 기아 완화: 특정 스레드가 자원을 무한정 기다리는 것을 방지하고, 대기 큐에서 벗어나 재시도하거나 다른 작업을 수행할 기회를 줍니다. 이는 불필요한 무기한 대기를 줄이는 데 도움이 될 수 있습니다.
- 시스템 반응성 향상: 특정 스레드가 자원을 얻지 못해 시스템 전체가 멈추는 교착 상태와 같은 상황을 예방하거나, 시스템이 '응답 없음' 상태에 빠지는 것을 막아 전반적인 시스템의 반응성을 높일 수 있습니다.

2)타임아웃 설정에는 중요한 한계점과 고려 사항
- 우선순위가 높은 태스크에 대한 잠재적 불이익: 만약 시스템 내에 높은 우선순위를 가진 태스크가 자원을 기다리는 상황에서 타임아웃으로 인해 대기 큐에서 강제로 제거된다면, 해당 태스크가 작업을 완료하는 데 필요한 자원을 제때 얻지 못해 시스템의 중요한 기능에 지연을 초래할 수 있습니다. 이는 전반적인 시스템의 효율성을 저해할 수 있습니다.
- 근본적인 기아 문제 해결 불가: 타임아웃은 기아 현상의 근본적인 원인인 불공정한 스케줄링 또는 자원 할당 정책을 직접적으로 해결하지 못합니다. 단순히 대기 시간을 제한할 뿐, 해당 스레드가 다시 자원을 얻을 수 있다는 보장은 없습니다. 기아 현상을 효과적으로 방지하기 위해서는 운영체제의 스케줄링 정책 예를 들어 FIFO, 우선순위 큐, 공정성 보장 메커니즘 등을 적절히 설계하는 것이 더 중요합니다.
- 스케줄러의 역할과의 중복 가능성: 현대 운영체제의 선점형 스케줄러는 이미 스레드의 우선순위와 예상 실행 시간 등을 고려하여 자원을 할당하고 스레드를 배치합니다. 이러한 환경에서는 wait() 연산에 별도의 타임아웃을 설정하는 것이 기아 현상을 완화하는 데 큰 영향을 미치지 않거나, 오히려 스케줄러의 판단과 충돌하여 비효율을 초래할 수도 있습니다.

따라서 타임아웃 설정은 교착 상태 예방이나 반응성 향상 등 일부 긍정적인 효과를 가져올 수 있지만, 기아 현상을 완전히 해결하는 만능 해결책은 아니며, 시스템의 특성과 스케줄링 정책을 종합적으로 고려하여 신중하게 적용해야 합니다.

Q. 2번째 파이팀의 질문 :
모니터 내에 접근하는 스레드들과 그 스레드 내부에서의 우선순위에 대해, 각 스레드마다 미리 우선순위와 작업 순서를 모니터 밖에서 정하고 들어오면 경쟁 상태 없이 오히려 더 효율적일 수 있을까요?

A. 멘토님의 답변 :
실제로 멀티코어 CPU에서는 비슷한 방식으로 CPU 코어가 각 필요한 부분에 맞게 패킷 일부를 나눠서 처리합니다. 그리고 각 코어는 독립적으로 각각 자료구조 인스턴스를 가지는데, 이렇게 가지는 것이 일관성 문제를 해결하여 공유된 캐시에 접근이 가능합니다. 하지만 이런 방법이 각 코어가 모든 자료구조 데이터를 각각 가지는 것이 비효율적이지 않은가 의심할 수 있습니다. 자료구조 데이터의 용량을 고려했을 때, 현재 방안이 제일 효율적이라고 볼 수 있습니다.

스레드들이 모니터 진입 전에 우선순위와 작업 순서를 미리 정하는 방식은 특정 상황에서 효율성을 높일 수 있지만, 일반적인 운영체제 환경에서는 오히려 비효율적일 가능성이 높습니다.

이 방식의 장점은 다음과 같습니다.
- 경쟁 상태 및 문맥 교환 오버헤드 감소: 스레드들이 정해진 순서대로 진입한다면, 임계 구역 내부에서의 자원 경쟁이 줄어들고, 불필요한 문맥 교환 오버헤드가 감소하여 효율성이 향상될 수 있습니다.
- 예측 가능한 성능: 특정 작업의 실행 순서가 매우 중요하거나, 예측 가능한 성능이 필수적인 실시간 시스템과 같은 제한된 환경에서는 이러한 사전 정의된 순서가 효율적인 접근 방식이 될 수 있습니다.
그러나 다음과 같은 단점 및 현실적인 어려움이 더 크게 작용합니다.
- 동적인 환경에서의 복잡성: 현대 운영체제 환경은 매우 동적입니다. 새로운 스레드가 생성되거나, 기존 스레드가 종료되거나, 심지어 우선순위가 변경되는 등 예측 불가능한 상황이 빈번하게 발생합니다. 이러한 동적인 변화에 맞춰 모든 스레드의 순서를 미리 정하고 유지 관리하는 것은 엄청나게 복잡하고 비효율적입니다. 모든 스레드의 예상 실행 시간과 의존 관계를 미리 정확히 파악하는 것은 거의 불가능합니다.
- 시스템 유연성 저하: 미리 순서를 정해놓으면 시스템의 유연성이 크게 떨어집니다. 긴급하거나 예상치 못한 작업이 발생했을 때, 미리 정해진 순서 때문에 지연되거나 적절하게 처리되지 못하는 문제가 생길 수 있습니다. 이는 시스템의 반응성을 저해하고 사용자 경험을 악화시킬 수 있습니다.
- 전역 동기화의 병목 현상: 모든 스레드의 순서를 미리 결정하고 이를 전역적으로 동기화하려면 추가적인 복잡한 동기화 메커니즘이 필요합니다. 이는 오히려 병목 현상을 유발하여 시스템의 확장성과 동시성을 저해할 수 있습니다. 모니터의

목적 중 하나는 임계 구역에 대한 접근을 지역화하여 동시성을 높이는 것인데, 전역적인 순서 지정은 이와 상반되는 결과를 초래할 수 있습니다.
- 단일 실패 지점 가능성: 미리 정해진 순서를 관리하는 로직에 문제가 발생하면 시스템 전체가 영향을 받을 수 있는 단일 실패 지점이 될 위험도 있습니다.

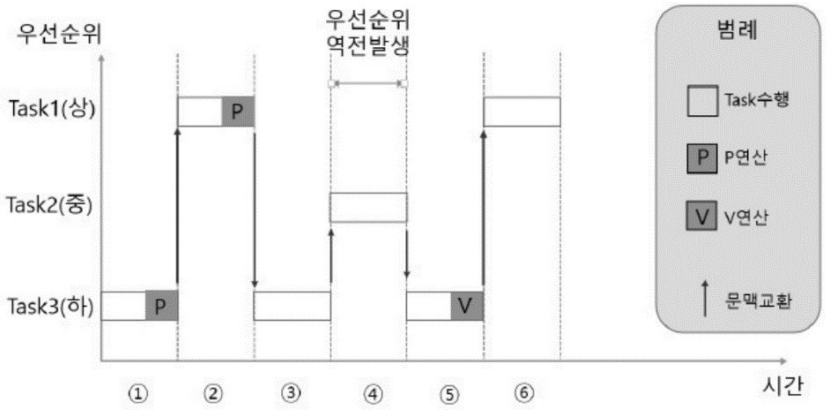
스레드 우선순위와 작업 순서를 미리 정하는 방식은 특정 제한적인 상황에서만 효율성을 보일 수 있으며, 범용적인 운영체제 환경에서는 그 복잡성과 유연성 저하로 인해 오버헤드가 더 커질 가능성이 높습니다. 대신, 모니터 내부에서는 mutex와 조건 변수를 통해 경쟁을 최소화하고, 운영체제 스케줄러가 우선순위 기반 스케줄링이나 시간 분할 등의 기법을 사용하여 스레드의 실행을 효율적으로 관리하는 것이 일반적이고 효과적입니다.

3. 우선순위 역전

실시간 운영 시스템에서 효율적인 작업 스케줄링은 매우 중요하다. 이러한 시스템에서는 프로세스의 우선순위가 시스템의 성능을 결정하는 결정적인 요소 중 하나이다. 그러나 때로는 우선순위 역전이라는 현상이 발생하여, 시스템의 성능과 신뢰성에 부정적인 영향을 미칠 수 있다. 우선순위 역전은 낮은 우선순위의 태스크가 고우선순위 태스크의 실행을 방해할 때 발생하며, 이는 예상치 못한 지연을 초래하고 전체 시스템의 안정성을 위협할 수 있다. 본 보고서에서는 우선순위 역전의 개념을 자세히 설명하고, 실제 사례를 통해 그 문제점을 분석한 후, 해결 방안을 제시하고자 한다.

3.1 우선순위 역전이란?

우선순위 역전은 높은 우선순위 작업이 낮은 우선순위 작업에 의해 차단되는 현상을 의미한다. 이는 두 작업이 공유 데이터를 액세스하려고 할 때 일반적으로 발생하며, 실시간 시스템의 스케줄링 가능성과 예측 가능성에 심각한 영향을 미친다.



위의 사례에서는 낮은 우선순위 T3가 수행되고, P연산을 수행한 후 임계 구역(critical section)을 점유한다. 이때 높은 우선순위 T1이 P연산을 시도하지만, T3의 임계 구역 선점으로 인해 차단 된다. 이후 T3의 수행이 재개되고, T1이 T2보다 우선순위가 높음에도 불구하고 T2가 먼저 수행된 다. 이는 우선순위 역전 현상 때문이다. T2의 수행이 종료되면 T3가 다시 수행되고, V연산을 통해 임계 구역이 해제된다. 마지막으로 T1이 수행된다. 이러한 우선순위 역전 현상은 우선순위 상속과 우선순위 천장 기법으로 해결이 가능하다.

3.2 우선순위 역전 원인

1) 비선점 가능한 코드 영역(non-preemptible regions of code):
- 코드의 특정 구역이 선점될 수 없는 경우, 높은 우선순위의 작업이 낮은 우선순위의 작업이 종료될 때까지 기다려야 할 수 있다.

2) 인터럽트(interrupt):
- 인터럽트는 시스템 자원을 강제로 점유할 수 있어 우선순위 역전을 초래할 수 있다.

3) 일부 작업에 대한 비유일한 우선순위(non-unique priorities for some tasks):
- 작업의 우선 순위가 고유하지 않거나, 우선순위 수준이 충분하지 않아 여러 작업이 같은 우선순위를 가 질 때, 우선순위 역전이 발생할 수 있다.

4) FIFO 큐(FIFO Queue):
- FIFO 큐는 작업이 도착한 순서대로 처리되어 우선순위가 높은 작업 이 낮은 우선순위 작업 뒤에 대기하게 될 수 있다.

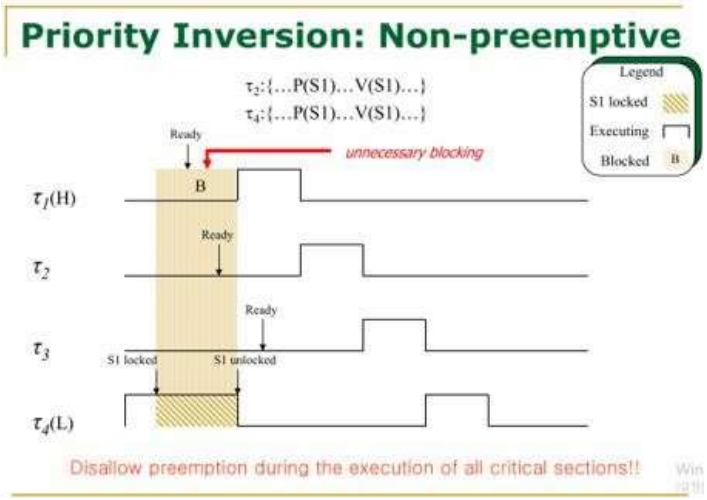
5) 동기화 및 상호 배제(synchronization and mutual exclusion):
- 동기화와 상호 배제 메커니즘 자체가 우선순위 역전을 유발할 수 있다. 예를 들어, 세마포어가 이미 낮은 우선순위의 작업에 의해 점유된 경우, 높은 우선순위의 작업은 세마포어가 해제될 때까지 대기해야 한다.

3.3 우선순위 역전 해결 방법

우선순위 역전 해결 방법에는 대표적으로 4가지가 있기에 이를 시간 순서로 알아보고자 한다. 초기 매커니즘으로는 비선점적 스케줄링이 등장하였다. 이어서 우선순위 상속 프로토콜과 우선 순위 천장 프로토콜이, 최종적으로는 최상위 잠금 우선순위 프로토콜에 대해 알아보고자 한다.

3.3.1 비선점적 스케줄링(Non-preemptive Scheduling)

비선점적 스케줄링은 크리티컬 섹션 동안 태스크가 선점되지 않도록 하여 우선순위 역전 현상을 방지하는 방법이다. 이 방법에서는 낮은 우선순위 태스크가 크리티컬 섹션에 들어가면 해당 태스크가 크리티컬 섹션을 벗어날 때까지 실행되며, 이 동안 다른 태스크, 심지어 높은 우선순위 태스크도 크리티컬 섹션을 선점할 수 없다. 크리티컬 섹션이 끝나면 태스크가 선점 가능해지며, 높은 우선순위 태스크가 실행될 수 있다.

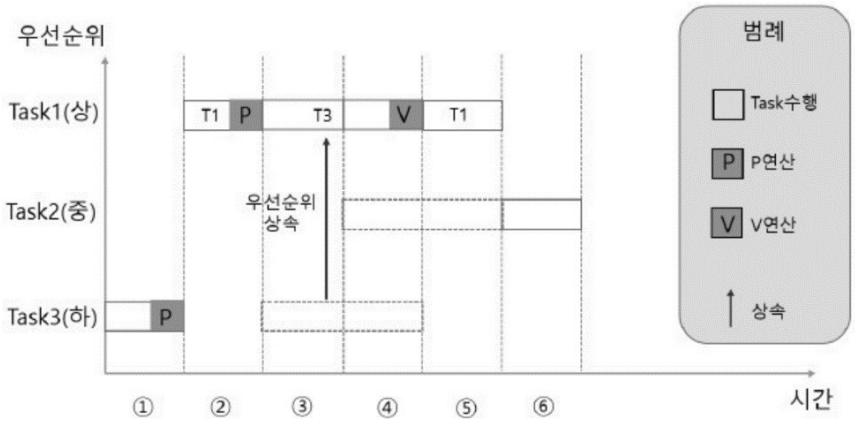


낮은 우선순위 태스크 τ4가 크리티컬 섹션에서 실행 중일 때, 높은 우선순위 태스크 τ1 이 그 크리티컬 섹션에 접근하려고 하면, τ4가 실행을 완료할 때까지 τ1는 블로킹된다. 이 는 중간 우선순위 태스크 τ2, τ3이 불필요하게 블로킹 되는 상황을 방지한다. 즉, 중간 우선 순위 태스크는 τ1이 τ4에 의해 블로킹되는 동안 계속 실행될 수 있다.

이러한 방식은 크리티컬 섹션 동안 선점을 허용하지 않음으로써 우선순위 역전 현상을 방지하고 시스템의 예측 가능성과 안정성을 유지할 수 있는 장점이 있다. 그러나 비선점적 스케줄링은 시스템의 전체 응답성을 저하시킬 수 있는데, 이는 모든 크리티컬 섹션이 실행 될 때까지 다른 태스크가 실행되지 않기 때문이다. 따라서 높은 우선순위 태스크가 낮은 우 선순위 태스크의 크리티컬 섹션 때문에 불필요하게 블로킹될 수 있는 단점이 존재한다.

3.3.2 우선순위 상속 프로토콜(PIP)

우선순위 상속 프로토콜은 다음과 같은 방식으로 작동한다. 작업 J가 하나 이상의 높은 우선순위 작업을 차단할 때, J는 자신의 원래 우선순위를 무시하며 차단한 모든 작업 중 가장 높은 우선순위 수준에서 자신의 중요한 섹션을 실행한다. 이는 낮은 우선순위 작업이 높은 우선순위를 임시로 상속할 수 있게 하여, 높은 우선순위 작업이 낮은 우선순위 작업에 의해 차단되는 시간을 줄인다.

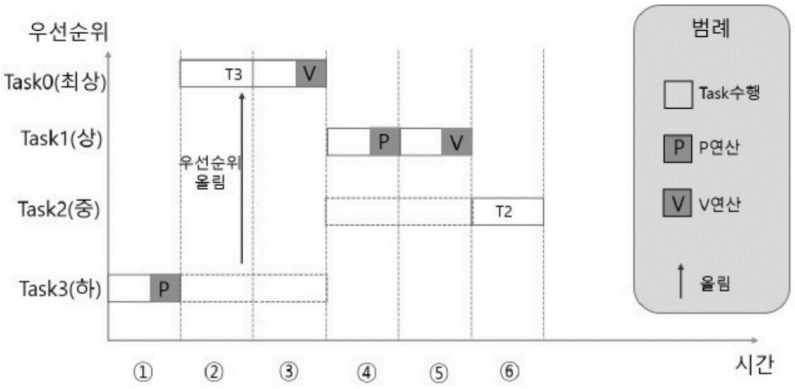


위의 사례에서는 낮은 우선순위인 T3가 수행되어 P연산을 수행한 후 임계 구역을 점유한다. 이때 높은 우선순위인 T1이 P연산을 시도하지만, T3의 임계 구역 선점으로 인해 차단된다. T1이 차단된 상태에서 T3의 우선순위가 T1의 레벨로 상승하여 T2보다 높은 우선순위 로 수행된다. 이후 V연산으로 임계 구역이 해제된다. 마지막으로 T1이 임계 구역을 점유하고 태스크를 수행하게 된다.

우선순위 상속 프로토콜은 낮은 우선순위를 높여준다. 이 방법의 장점은 자동으로 우선 순위가 동적으로 변경된다는 점이다. 그러나 단점으로는 우선순위 변경에 따른 오버헤드가 발생한다는 점이 있다. 또한, 교착상태가 발생할 가능성이 존재하며, 자원 동시 소유는 처리 불가능하다.

3.3.3 우선순위 천장 프로토콜(PCP)

우선순위 천장 프로토콜의 목표는 교착 상태와 연쇄 차단의 형성을 방지하는 것이다. 세마포어 S의 우선순위 천장은 S를 잠글 수 있는 가장 높은 우선순위 작업의 우선순위이다. 시스템 천장은 현재 다른 작업이 잠근 모든 세마포어의 최대 우선순위 천장이다.

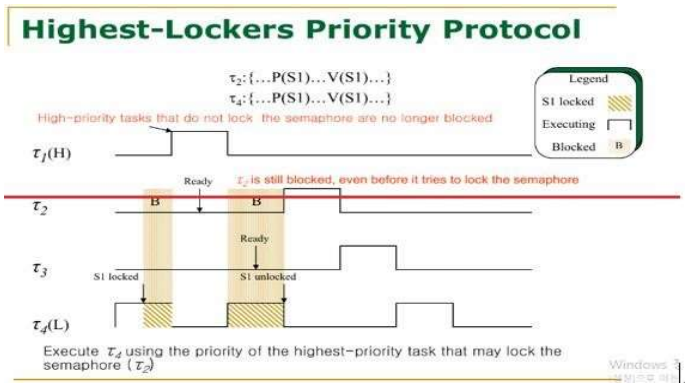


위의 사례에서는 낮은 우선순위인 T3가 수행되고, P연산을 수행한 후 임계 구역을 점유 한다. 이후 우선순위가 최우선순위로 올려지면서 T1은 T3보다 우선순위가 낮아 대기하게 된다. T3가 종료되면 T1이 수행되고, T1이 종료된 후 T2가 수행된다. 이는 T2의 우선순위가 가장 낮기 때문이다.

우선순위 천장 프로토콜은 우선순위를 한정한다. 이 방법은 교착상태가 발생하지 않으며, 자원 동시소유 처리가 가능하다. 우선순위 변동에 대한 부하가 적용된다는 장점이 있지만, 우선순위 한정범위 설정에 어려움이 있다는 단점이 있다.

3.3.4 최상위 잠금 우선순위 프로토콜 (Highest-Lockers Priority Protocol, HLPP)

최상위 잠금 우선순위 프로토콜은 자원을 잠그는 태스크 중에서 가장 높은 우선순위를 가진 태스크의 우선순위를 상속받아 실행하는 방식으로, 높은 우선순위 태스크가 자원을 잠그지 않은 경우에도 블로킹 되지 않도록 보장하는 방법이다. 이 프로토콜은 낮은 우선순위 태스크가 공유 자원을 잠글 때, 중간 또는 높은 우선순위 태스크가 자원을 요청하면 해당 태스크가 가장 높은 우선순위를 상속받아 크리티컬 섹션을 빠르게 완료하고 자원을 해제한다.



위 그림은 높은 우선순위 잠금 프로토콜의 일종으로, PIP의 변형된 형태를 보여준다. 낮은 우선순위 태스크 τ4가 S1을 잠금으로써, τ2와 τ3은 S1을 잠그려고 하지만 τ4가 이미 잠금 중이므로 블로킹 된다. τ4는 높은 우선순위 태스크 τ2의

우선순위를 상속받아 실행되며, 이로 인해 τ4가 빨리 완료되어 S1을 해제할 수 있게 된다. 이러한 특징 덕분에 높은 우선순 위 태스크가 자원을 잠그지 않은 경우 블로킹되지 않으며, 높은 우선순위 태스크가 더 빨리 실행될 수 있도록 보장한다.
이를 통해 높은 우선순위 태스크의 블로킹을 최소화하고 자원 사용의 효율성을 높이며, 우선순위 역전 현상을 완화하여 시스템의 예측 가능성과 안정성을 유지할 수 있다. 그러나 우선순위 상속 과정의 복잡성 증가와 추가적인 시스템 오버헤드, 모든 우선순위 역전 현상을 완벽히 해결하지 못할 수 있는 단점이 존재한다.

3.4 결론

우선순위 역전은 실시간 시스템에서 중대한 성능 저하를 초래할 수 있는 중요한 문제이다. 이를 해결하기 위해 우선순위 상속과 우선순위 천장 프로토콜과 같은 다양한 기법이 개발되어 왔으며, 이러한 기법들은 실제 산업 현장에서 널리 적용되고 있다. 실시간 시스템의 설계 및 운영에 있어서 우선순위 역전 문제를 고려하는 것은 시스템의 안정성과 신뢰성을 보장하는 데 필수적인 요소이다. 향후 연구에서는 이 문제에 대한 더욱 효율적이고 혁신적인 해결 방안이 모색될 것이라고 본다.

3.5 멘토님과의 튜터링

Q. 파이팀의 질문 :

PCP의 경우에 context-switch가 자주 발생해 오버헤드를 유발한다고 볼 수 있을까요?

A. 멘토님의 답변 :

아닙니다. 오히려 PCP를 사용하게 되면 Context Switch는 적게 발생할 가능성이 높습니다. 교재에서 설명하는 실시간 스케줄의 Timeline은 일반적으로 CPU 코어가 1개일 때를 가정하여 설명하고 있습니다. 이는 우선순위가 다른 여러 Task (또는 Job)이 있고, 공유 자원에 대해 Lock/Unlock을 하는 상황에서 스케줄이 어떻게 되는지에 대한 이해를 돕기 위함입니다.
PCP의 핵심은 우선순위 역전 현상을 방지하는 데 있습니다. 이를 위해 PCP는 공유 자원에 접근하려는 Task의 우선순위를 일시적으로 높여주거나, 해당 자원을 이미 점유하고 있는 Task의 우선순위를 시스템 내에서 가장 높은 우선순위로 설정하는 System Ceiling 개념을 도입합니다.
이러한 System Ceiling이 설정되면, 우선순위가 낮은 Task가 잡은 Lock으로 인해 System Ceiling이 높아지고, 그로 인해 더 높은 우선순위의 Task가 깨어나더라도

(or Ready 상태가 되더라도) 즉시 Context Switch가 발생하지 않을 수 있습니다. 높은 우선순위의 Task는 시스템 실링보다 낮은 우선순위를 가지므로, 현재 실행 중인 낮은 우선순위의 Task가 Lock을 해제할 때까지 기다리게 됩니다.
결과적으로, 불필요한 Context Switch를 줄여 시스템 오버헤드를 감소시키는 효과를 가져옵니다. Context Switch는 그 자체로 CPU 시간과 자원을 소모하는 작업이므로, 이를 최소화하는 것은 실시간 시스템의 성능에 매우 중요합니다.

4. 참조문헌

- Operating System Concepts, 10th, Silberschatz 외 2명
- Linux Kernel Archives Documentation on CFS
- Silberschatz et al., Operating System Concepts
- 이화영, '운영체제', 한빛아카데미
- A. S. Tanenbaum, Modern Operating Systems, Pearson
- William Stallings, Operating Systems: Internals and Design Principles, Pearson
- Andrew S. Tanenbaum, Herbert Bos, Operating Systems: Design and Implementation, Pearson
- L. Sha et. al., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", IEEE Transactions on Computers, 39(9), 1990
- 송재신, 심재홍, 최경희, 정기현, 김홍남. 우선순위 역전 문제를 해결하기 위한 통합 실시간 스케줄링 모델, 한국통신학회논문지 '01-7 Vol.26 No.7A', P1-12