운영체제 8장
Deadlocks
서로가 서로를 기다려서 프로그램 진행이 더 이상 안 되는 상태
0. System Model
여러 proc가 자원을 concurrency해서 자원을 공유하는 모델
1. Deadlock in Multithreaded Applications
멀티쓰레드에서 Deadlock이 발생하는 상황
2. Deadlock Characterization
Deadlock이 무엇이고, 언제 발생하는가?(발생 조건)
3. Methods for Handling Deadlocks
DeadLock 해결 방법
4 가지(발생 조건도 4가지인데, 그 중 하나라도 해결하면 발생 X)
4. Deadlock Prevention
5. Deadlock Avoidance
H/W만 사용한 해결 방법 이용 어 려움으로 인해, OS나 라이브러리 API에서 H/W 기능을 이용해 프로그래머가 쉽게 동기화 구현 가능하도록 만듦
6. Deadlock Detection
일종의 M/W, Programming에서 자동으로 Critical-Section 보호할 수 있는 쉬운방법 제공
7. Recovery from Deadlock

목표
-mutex locks 사용하다가 Deadlock 발생 가능
-Deadlock이 발생할 수 있는 네 가지 조건
-자원 할당 그래프(allocation graph)로 Deadlock 상황 표현 가능
-Deadlock이 발생하는 네 가지 조건과 preventing 방법(네 가지 조건 중 하나만 상쇄하면..)
-banker's algorithm for deadlock avoidance
자원을 할당할 때, Deadlock 발생 가능하면 할당 자체를 X (사전 방지)
-Deadlock detection 알고리즘
-Deadlock으로부터 recovering하는 접근법

0. System Model
proc가 자원 사용하는 것을 formal하게 modeling할 필요성 존재(Deadlock, C.S 등 때문에)

(1) Resource type 정의 (R1, R2, ... , Rm)
-OS가 여러 가지 종류의 자원을 관리함 ex. CPU cycle, memory space, I/O device
(자원 type R에 W개의 자원 객체 존재 가능)
-proc가 자원 사용 시 : request, use, release 과정 거침

1. Deadlock in Multithreaded Applications
-초기에 두 개의 mutex locks 생성함

-Deadlock 예시 코드(8.6p)

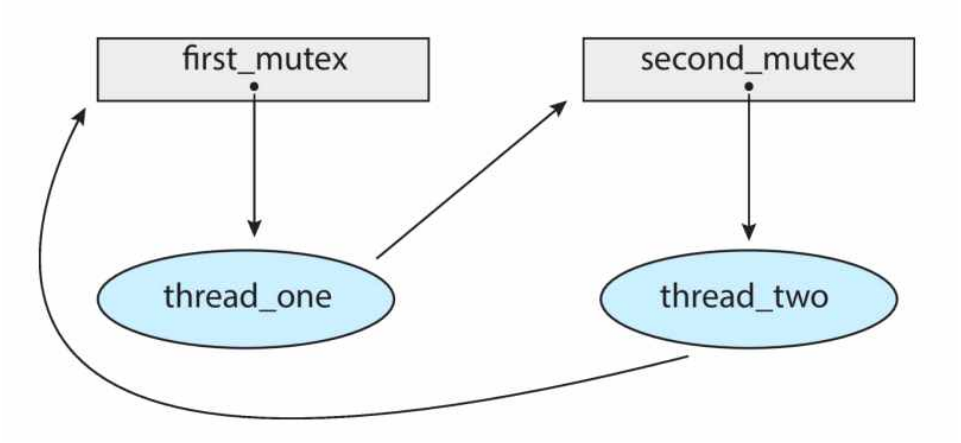


그림 1. Resource Allocation graph

2. Deadlock Characterization

(1) Deadlock 발생 조건(4 가지 조건 모두 만족해야 Deadlock 발생)

-Mutual exclusion(상호 배제, 독점권)

-Hold and wait

자기가 자원을 가지고 있고, 또 다른 자원을 원하고 있을 때(얻지는 못한 상태)

-No preemption(자원을 강제로 빼앗을 수 없음)

-Circular wait(Hold and wait랑 유사, circle 형성)

자신이 자원을 잡고 다른 proc가 잡고 있는 자원을 대기하는데, proc 순차적으로 대기 상태가 이어지는 경우

(2) Resource-Allocation Graph

Process와 Resource 간의 관계에 대해 표현한 것

-구성(P, R은 node 역할이고, request, asssignment는 edge 역할)

P = {P1, P2, ... Pn}

R = {R1, R2, ... Rm}

request edge : proc가 자원에게 요청

assignment edge : proc가 해당 자원의 사용권을 얻었음

-장점

Graph를 통해 어느 proc와 resource가 존재하고 어느 proc가 resource를 사용하고, request를 하고 있는지 쉽게 표현하고 파악할 수 있음

-Basic Facts (Deadlock 판단 방법)

> Graph에서 no cycles → no deadlock

> Graph에 서 contains a cycle

→ 각 Resource 타입마다 하나의 객체 있을 경우, deadlock

→ 각 Resource 타입마다 여러 객체 있을 경우, deadlock 가능성 존재

(Resource 집합 내 객체들이 모두 Circle과 연관되어 있어야 deadlock)

3. Methods for Handling Deadlocks
DeadLock 해결 방법
① Deadlock Prevention
② Deadlock Avoidance
자원 할당 시, Deadlock 발생 가능하다고 판단하면 할당 자체 안함
③ Detection and Recovery
④ DeadLock 발생하든 말든 무시하는 것(이때, 사용자가 시스템 reboot 등 수동 해결)

✓ 1, 2번의 경우 DeadLock이 아예 발생하지 않음(알고리즘 복잡, 자원 활용도 ↓)
✓ 따라서 DeadLock 발생을 허용하되(3번), DeadLock을 탐지하고 벗어나도록 함

4. Deadlock Prevention
Deadlock 발생 조건 4 가지 중 하나만 없애면 Deadlock 해결 가능

#조건 및 해결 방법(4 가지 모두 쉽지 않고, 거의 모두 자원 활용도를 낮추는 방향)
Mutual Exclusion
▶프로그램 진행 및 C.S 보호에서 필수적이므로 없애기 힘듦(해결 거의 불가)
Hold and Wait
▶program 수행 시, 모든 자원을 한꺼번에 할당시키고 종료 시 한 번에 release하는 방식
(자원 사용에 너무 낭비가 심해 자원 활용도 ↓, Starvation 가능성)
No Preemption
▶한 proc가 자원을 잡지 못하면, 그 전에 잡아놓았던 자원까지 놓게 하는 것
(새로운 자원을 잡지 못한다고 그 전 잡았던 자원까지 버리기 때문에 자원 활용도 ↓)
Circular Wait
▶자원마다 번호 매겨, 어떤 proc가 자원 사용 시 저→고 번호 순서로 자원 잡도록 함
(모든 자원에 번호 매기고 순서대로 잡도록 하는 것은 자원 사용 어렵고 자원 활용도 ↓)

5. Deadlock Avoidance
전제 조건 : 어떤 proc가 어떤 자원을 Maximum 몇 개 필요로 하는지 알고 있어야 함

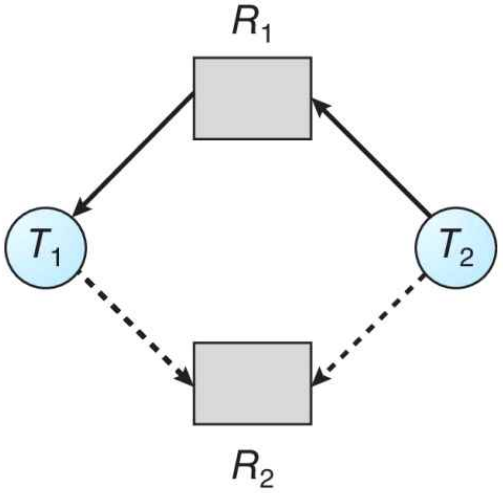
-방식
자원 할당해 줄 때, 자원이 Circular wait가 되지 않도록 설정
(Deadlock 발생 시나리오가 있는지 검토하여 가능성 있다면 할당 자체 X)

-Safe State(Deadlock 발생할 상황인지 아닌지 판단하기 위해 이용)
Deadlock이 발생하지 않은 상태를 말하며, 모든 Proc가 만족스럽게 끝날 수 있는 sequence를 지칭하기도 함(↔Unsafe State)

✓ Deadlock Avoidance는 Safe인지 Unsafe(deadlock 가능)인지 확인하여 자원 할당 결정
- Avodiance 알고리즘

Resource 타입 당 하나의 객체 있을 시 → resource-allocation graph 체크
Resource 타입 당 여러 객체 있을 시 → graph의 circle로 판별 불가하여, Banker's Algorithm 사용

(1) Resource-Allocation Graph Scheme
Node 종류 : proc, resource
Edge 종류 : Claim Edge(=request Edgem, A->B A가 B 자원을 사용하고 싶다),
Assignment Edge(A<-B A가 B자원을 사용하고 있다)



(2) Banker's Algorithm
자원이 여러 개 존재할 때, Deadlock 예방하는 방법(cycle 형성해도, Deadlock 아닐 수 O)
> Banker's 알고리즘을 통해 Deadlock인지 아닌지 판단
전제 조건: 어느 proc가 어느 자원을 최대 몇 개까지 사용한다는 것을 알고 있어야 함

#Banker's 알고리즘에서 사용하는 데이터 구조
-n : 프로세스 개수, m : 자원 타입 개수
Available : 현재 가지고 있는(남아있는) 자원 개수 <available[j] = k, j번째 자원이 k개 남아 있음>
Max : n x m 행렬, Max[i,j] = k 는 i가 j번째 자원을 k개까지 필요하다.
Allocation : n x m 행렬, Allocation[i,j] = k 는 I가 j번째 자원을 k개 사용 중이다.
Need : n x m 행렬, Need[i,j] = k 는 i가 j번째 자원을 앞으로 k개 더 필요하다.
✓ Need[i,j] = Max[i,j] - Allocation[i,j]

(3) Safety Algorithm
-Work : 시스템 운영체제가 가지고 있는 자원의 개수
-Finish : 어떤 proc가 자신 수행을 끝냈는지
단계
A. **Initialize:**
Work = Available (처음이니 모든 자원 사용 가능)

Finish [i] = false for i = 0, 1, …, n- 1 (아무도 끝나지 않음)

B. 다음 조건을 만족하는 i 찾기
-Finish [i] = false
-Need(i), i가 필요한 배열 내 자원 값 <= Work
만족하는 i가 없다면, D 단계로 넘어감

C. Work = Work + Allocation(i)
Finish[i] = true (자원 사용을 다 끝내고 종료한다는 것)
i proc는 종료되었고 b 단계로 가서 다음 만족하는 proc 탐색 진행

D. 만족하는 모든 i에 대한 Finish [i] == true라면, 시스템이 safe state라고 판단함

(4) 어떤 Proc가 자원을 요청했을 때, 요청하는 것에 대해 판단하는 과정
A. Request(i) <= Need(i) 확인하고(필요한 것 이상으로 요청하면 maximum claim 초과할 수 있음), B 단계로 진행

B. Request(i) <= Available 확인하고, C 단계로 진행 (요청을 들어주려면 남아있는 자원이 더 많아야 함)

C. 자원 할당 시 안전한지 판단하기 위해 할당하는 척함(자원을 곧바로 할당하는 것 X)
* 방법 - modifying the state as follows:
Available = Available – Request(i);
Allocation(i) = Allocation(i) + Request(i);
Need(i) = Need(i) – Request(i);
>> 이후, safe인지 unsafe한지 확인하여 safe하다면 할당 진행

6. Deadlock Detection
Deadlock Prevention, Deadlock Avoidance에서는 Deadlock이 발생하지 않으나, 알고리즘이 복잡하고 자원 사용의 낭비가 심해 범용적 사용이 어려움.
✓ 따라서 Deadlock 발생 허용하고, Recovery 하는 것이 현실적임

(1) Single Instance 일 때, Deadlock Detection 하는 방법
wait-for graph(=resource allocation graph와 유사) 이용, circle 형성되어있는지 확인
P(i) -> P(j), 프로세스 I는 프로세스 j가 사용하는 자원을 기다린다는 의미
여기서 circle 형성되어있으면, Deadlock으로 탐지함
(Deadlock 탐지 연산량은 프로세스 개수가 n일 때, n^2)

(2) Several Instances 일 때, Deadlock Detection 하는 방법
circle 형성되어 있더라도, Deadlock 발생하지 않는 상황 존재함
✓ 따라서, 다른 알고리즘 사용(Banker's 알고리즘, Safety 알고리즘과 유사)
-기본 개념
Available : 현재 가지고 있는(남아있는) 자원 개수
Allocation : n x m 행렬, Allocation[i,j] = k 는 I가 j번째 자원을 k개 사용 중이다.
Request : n x m 행렬, Need[i,j] = k 는 i가 j번째 자원을 k개 요청하고 있다.

-진행 단계
A. Initialize:
Work = Available
For i = 1,2, …, n에 대해 만약 Allocation(i) != 0라면, Finish[i] = false; 하고 그렇지 않다면, Finish[i] = true (끝나지는 않았으나 자신이 자원을 할당해줄 수 있는 proc 탐색)

B. 두 조건을 만족하는 i 탐색
(a) Finish[i] == false
(b) Requesti ⓞ Work
만약 만족하는 i 없다면, D 단계로 넘어감

C. 지금 당장은 request한 자원 주어 자원 개수 줄어듦
(But, proc 종료 시 새로 할당해준 request 자원 반납 + 원래 가지고 있는 자원 반납)
Work = Work + Allocation(i) (따라서 증가하는 효과)
Finish[i] = true (proc가 종료가 되었음을 의미)
다시 B 단계로 가서 탐색 진행
(모든 proc가 true로 끝난다면 deadlock이 발생하지 않는다는 것)

D. 만약 Finish[i] == false인 것이 존재한다면, 그 proc I가 deadlock에 연관되어 있는 것
-n : 프로세스 개수, m : 자원 개수
알고리즘 복잡도 : O(m x n^2)
whether the system is in deadlocked state

-Detection 알고리즘의 사용 시 중요점
알고리즘 복잡도가 높아, proc와 자원의 수가 적다면 문제 없지만 많은 경우 부담됨
▶ Detection 알고리즘을 어느 주기로 돌려야 하는지? (Deadlock 발생 주기 판단 필요)
▶ proc가 자원을 갖고 진행하다 deadlock detection을 통해 탐지되면 자원을 되돌려 놓으면서 진행 상황도 되돌리는데, 얼마나 많은 프로세스가 진행 상황을 되돌려야 하는지?

7. Recovery from Deadlock
deadlock을 탐지하면, deadlock에서 복구하는 작업이 필요함
ex. deadlock 연관 proc 종료, deadlock 연관 proc 진행 상황 되돌림(roll back)
어떤 proc를 Abort 시키냐? 하는 기준 존재(우선순위, proc 길이 등등)

-Recovery 작업 중 여러 가지 결정해야 하는 사항
 victim 설정(Abort, roll back해야 하는 proc) - 비용 최소화 방향으로
 Rollback(proc의 어느 일정 시점까지만 되돌리기)
 Starvation(deadlock이 너무 자주 발생해 어느 한 proc 계속 victim 될 수 있음) 고려