운영체제: 강의노트 09

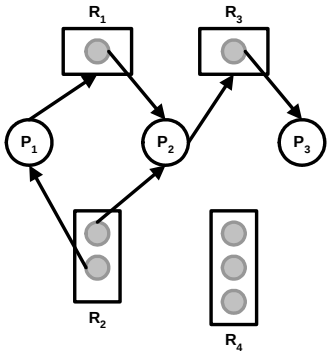
A. Silberschatz, P.B. Galvin, G. Gagne
Operating System Concepts,
Sixth Edition, John Wiley & Sons, 2003.

Part II. Process Management

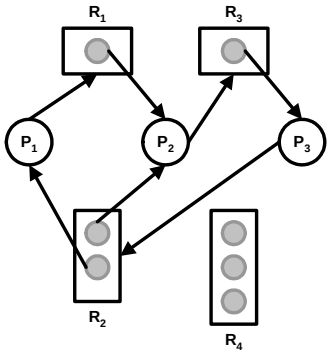
8 교착상태

8.1 시스템 모델

- 시스템 구성 요소
  - 자원
    - 유한 개의 자원이 존재하며, 이 자원은 몇 가지 유형으로 분류된다. 같은 유형의 자원이 여러 개 있을 수 있다.
    - 예) 주기억장치 공간, CPU 시간, 파일, 입출력 장치
  - 프로세스: 여러 프로세스가 여러 자원에 대해 경쟁한다.
- 프로세스가 유형 A의 자원을 요청하면 그 유형의 어떤 자원을 할당하여도 그 요청은 만족된다.
- 프로세스가 요청할 수 있는 자원의 수는 제한이 없으나, 시스템에 존재하는 자원보다 많은 수의 자원을 요청할 수 없다. 예) 이 시스템에 프린터가 두 개 있으면 프로세스는 두 개이하로만 요청할 수 있다.
- 프로세스는 자원을 사용하기 전에 요청을 해야 하며, 사용이 끝나면 해제해야 한다. 프로세스는 다음 순서로 자원을 이용한다.
  - 단계 1. **요청(request)**: 바로 요청을 허용할 수 없으면 요청한 프로세스는 기다려야 한다.
  - 단계 2. **사용(use)**
  - 단계 3. **해제(release)**: 획득한 자원의 사용이 끝나면 다른 프로세스들이 사용할 수 있도록 해제한다.
- 자원의 요청과 해제는 시스템 호출을 이용한다. 예) open, close
- 자원에 대한 대기는 세마포어를 이용하여 구현할 수 있다.
- 어떤 집합 내에 있는 모든 프로세스가 대기 상태이며, 이 집합 내에 있는 각 프로세스가 이 집합 내에 다른 프로세스가 가지고 있는 자원을 기다리고 있으면 교착상태에 있다고 한다.



<그림 8.1> 자원 할당 그래프



<그림 8.2> 교착상태가 있는 자원 할당 그래프

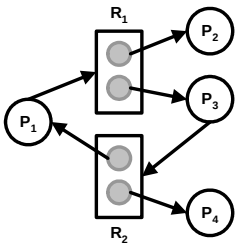
8.2 교착상태의 특징

8.2.1 필요조건

- 다음 네 개의 요구조건이 모두 충족되어야 교착상태가 발생한다.
  - **상호배제**: 최소한 하나는 비공유 방식으로 점유되어야 한다. 비공유 방식의 점유란 한 번에 하나의 프로세스만 자원을 사용할 수 있음을 말한다.
  - **점유와 대기**: 프로세스는 최소한 하나의 자원을 점유하고 있고, 다른 프로세스가 점유하고 있는 다른 프로세스를 기다리고 있어야 한다.
  - **비선점**: 점유된 자원은 강제로 해제될 수 없고, 프로세스가 자원의 사용을 끝마치고 자발적으로 해제할 때까지는 그 자원을 얻을 수 없어야 한다.
  - **순환 대기**: 다음을 만족하는 대기하는 프로세스의 집합 $\{P_0, P_1, \ldots, P_n\}$이 존재해야 한다. P_0는 P_1이 점유한 자원을 기다리고 있고, P_1은 P_2가 점유한 자원을 기다리고 있고, P_n은 P_0가 점유한 자원을 기다리고 있다.

8.2.2 자원 할당 그래프

- 자원 할당 그래프의 구성요소



<그림 8.3> 주기는 있지만 교착상태가 없는 자원 할당 그래프

  - 노드
    - 프로세스 노드: $P = \{P_1, P_2, \ldots, P_n\}$, 원으로 표시
    - 자원 노드: $R = \{R_1, R_2, \ldots, R_m\}$, 직사각형으로 표시하며, 직사각형 내부에 점으로 같은 유형의 자원의 인스턴스를 나타낸다.
  - 선
    - $P_i \rightarrow R_j$: **요청선**(requesting edge)으로 P_i가 R_j를 요청하였음을 나타낸다.
    - $R_j \rightarrow P_i$: **할당선**(assignment edge)으로 R_j가 P_i에 할당되었음을 나타낸다.

    요청선은 프로세스 노드에서 자원 노드를 나타내는 직사각형을 가리키는 반면에 할당선은 프로세스 노드에서 자원 노드를 나타내는 직사각형 내부에 있는 점과 연결된다.
- 예) 그림 8.1 참조.
- 주어진 자원 할당 그래프에 주기(cycle)가 없으면 교착상태가 없는 것이고, 있으면 교착상태가 존재할 수 있다. 그림 8.2와 그림 8.3 참조.
- 만약 각 자원이 하나의 인스턴스만 가지고 있으면 주기는 교착상태의 필요충분조건이다.

8.3 교착상태를 처리하는 방법

- 교착상태를 처리하는 세 가지 방법
  - 프로토콜을 사용하여 교착상태가 미리 일어나지 않도록 하는 방법.
  - 교착상태를 허용하지만 주기적으로 검사하여 교착상태를 제거하는 방법.
  - 교착상태를 무시하는 방법. 예) 유닉스
- **교착상태 예방(prevention) 방법**: 교착상태의 네 가지 필요 조건 중 한 가지가 발생하지 않도록 하여 예방한다. 이 방법은 자원 요청을 제한하여 교착상태를 예방한다.
- **교착상태 회피(avoidance) 방법**: 운영체제가 사전에 프로세스가 어떤 자원들을 요청할지 알아야 한다. 운영체제는 이 정보를 이용하여 프로세스가 자원을 요청하였을 때 이것을 허용할지 여부를 결정한다.
- **교착상태 탐지(detection) 방법**: 교착상태 예방 또는 회피 알고리즘을 사용하지 않으면 교착상태가 발생할 수 있다. 이 경우 교착상태를 발견하는 알고리즘과 교착상태로부터 시스템을 복구하는 알고리즘을 제공하여 교착상태 문제를 해결할 수 있다.
- 교착상태를 무시할 경우에는 결국에는 시스템을 재시작해야 한다.

8.4 교착상태 예방

8.4.1 상호배제

- 상호배제 조건을 통해 교착상태를 예방하는 것은 어렵다. 어떤 자원은 근본적으로 공유를 할 수 없다.

8.4.2 점유와 대기

- 한 자원을 점유하면서 다른 자원을 대기할 수 없도록 하는 두 가지 방법
  - 실행되기 전에 필요한 모든 자원을 점유하도록 하는 방법
  - 한 자원을 점유한 상태에서 다른 자원을 요청할 수 없도록 하는 방법
- 문제점
  - 할당받은 자원을 오래 동안 사용하지 않을 수 있으므로 자원의 사용 효율이 낮다.
  - 여러 자원이 필요한 프로세스는 영구구적으로 기다릴 수 있다.

8.4.3 비선점

- 사용할 수 있는 방법
  - 방법 1. 프로세스가 자원을 점유하고 있으면서 바로 할당될 수 없는 자원을 요청하면 그 프로세스가 점유하고 있는 모든 자원을 해제하는 방법.
  - 방법 2. 프로세스 P_i가 어떤 자원 R_i를 요청하였을 때 R_i를 또 다른 자원 R_j를 기다리고 있는 프로세스 P_j가 점유하고 있으면 R_i를 P_j로부터 해제하고 그것을 P_i에게 할당하는 방법

8.4.4 순환 대기

- 순환 대기가 일어나지 않도록 하는 방법: 각 자원에 번호를 부여하고 낮은 번호의 자원부터 할당을 받도록 한다.
  - $R = \{R_1, R_2, \ldots, R_m\}$이 자원의 집합이면 $F : R \rightarrow N$ 함수를 정의하여 각 자원의 유

형별로 번호를 할당한다. 예)

$$\begin{aligned} F(\text{tape drive}) &= 1 \\ F(\text{disk drive}) &= 5 \\ F(\text{printer drive}) &= 12 \end{aligned}$$

  - 방법 1. 프로세스 P_i가 R_i 자원을 점유하고 있으면 이 프로세스는 $F(R_j) > F(R_i)$인 자원만 요청할 수 있다.
  - 방법 2. 프로세스 P_i가 R_j 자원을 요청하기 전에 이 프로세스는 $F(R_j) < F(R_i)$인 모든 자원 R_i를 해제해야 한다.

- 방법의 정확성에 관한 증명(모순에 의한 증명)
  - $\{P_0, P_1, \ldots, P_n\}$은 순환 대기하는 프로세들의 집합이라 하자. 즉, P_i는 P_{i+1}이 점유하고 있는 R_i를 요청하는 프로세스이다.
  - 즉, P_{i+1}은 R_i를 점유하고 있으며 R_{i+1}을 요청하고 있다. 이 때 $F(R_i) < F(R_{i+1})$이다.
  - 그런데 $F(R_0) < F(R_1) < \cdots < F(R_n) < F(R_0)$이므로 $F(R_0) < F(R_0)$이다. 따라서 $\{P_0, P_1, \ldots, P_n\}$이 순환 대기하는 프로세들의 집합이라는 것은 모순이다.

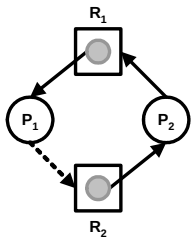
8.5 교착상태 회피

- 교착상태 회피 방법은 사전에 프로세스가 필요한 자원의 정보를 이용하여 그 프로세스들이 자원을 요청하였을 때 교착상태가 발생하지 않도록 지연하거나 할당해주는 방법이다.
- 회피 방법에서 프로세스가 자원을 요청하면 1) 현재 사용 가능한 자원, 2) 각 프로세스에게 할당된 자원, 3) 각 프로세스의 미래 요청과 해제를 고려하여 할당 또는 지연을 결정한다.

8.5.1 안전한 상태

- 일반적으로 교착상태 회피 알고리즘은 **안전한 상태(safe state)**를 정의하고, 프로세스가 자원을 요청하면 시스템이 안전한 상태(교착상태가 발생할 수 없는 상태)를 유지하는지 검사한 다음에 할당해준다.
- 이를 위해 각 프로세스는 미리 각 유형의 자원마다 필요한 최대 수를 선언해주어야 한다.
- 시스템에 **안전한 순서(safe sequence)**가 존재하면 그 시스템은 안전하다고 한다.
- 현재 할당 순서에 대해 다음을 만족하는 프로세스의 순서 $< P_1, P_2, \ldots, P_n >$이 존재하면 이 순서를 안전한 순서라 한다.

  각 P_i에 대해 P_i의 요청이 현재 사용 가능한 자원과 $j < i$인 P_j가 점유하고 있는 자원에 의해 만족될 수 있어야 한다.



<그림 8.4> 자원 할당 그래프 알고리즘의 예

- 즉, P_i는 자원을 할당받아 사용하거나 P_j가 모두 끝난 다음에 사용할 수 있다.
- 시스템이 안전한 상태에 있으면 교착상태가 발생하지 않는다. 시스템이 안전하지 않은 상태에 있으면 반드시 교착상태가 발생하는 것은 아니지만 발생할 수 있다.
- 예) 시스템에 12개의 테이프 드라이브가 존재하며, t_0 시간 때 다음과 같은 프로세스가 있다.

	최대 수요	현재 수요
P_0	10	5
P_1	4	2
P_2	9	2

  $< P_1, P_0, P_2 >$ 순서는 안전한 순서이므로 이 시스템은 현재 안전한 상태에 있다. t_1 시간에 P_2에게 하나 테이프가 더 할당되었다고 하자. 그러면 시스템은 안전하지 못한 상태가 된다. P_1은 할당되어 실행될 수 있지만 이 프로세스가 끝난 후에 P_0와 P_2가 필요한 최대만큼 요청하면 교착상태가 된다.

8.5.2 자원 할당 그래프 알고리즘

- 각 유형의 자원마다 하나의 인스턴스가 존재하면 사용할 수 있는 알고리즘
- 기존 자원 할당 그래프에 점선으로 표시하는 **요청가능선(claim edge)**을 추가한다.
  - $P_i \dashrightarrow R_j$: 미래에 P_i가 R_j를 요청할 수 있다는 것을 나타낸다.
- P_i가 R_j를 요청하면 요청선 $P_i \dashrightarrow R_j$을 할당선 $R_j \rightarrow P_i$로 바꾸어 주기가 없는지 검사하여 없는 경우에만 요청을 허용한다. 이 주기를 검사할 때 다른 프로세스의 요청가능선도 함께 고려된다.

8.5.3 은행가 알고리즘

- 일반적인 경우에 사용할 수 있는 교착상태 회피 알고리즘
- 각 프로세스는 생성되기 전에 각 자원의 유형마다 필요한 최대 인스턴스 수를 선언해야 한다. 이 수는 시스템의 전체 자원의 수를 초과할 수 없다.

- 프로세스가 자원을 요청하면 시스템이 안전한 상태를 유지하는지 검사한다.
- 시스템에서 유지하는 데이터 구조: n이 프로세스의 수이고, m이 자원의 유형 수이다.
  - 사용 가능(*avail*): 길이 m인 벡터로 각 자원의 유형마다 사용 가능한 자원의 수를 나타낸다.
  - 최대 수요(*max*): $n \times m$ 행렬로 각 프로세스의 각 자원에 대한 최대 수요를 나타낸다.
  - 할당(*alloc*): $n \times m$ 행렬로 각 프로세스에게 할당되어 있는 각 자원의 인스턴스 수를 나타낸다.
  - 남은 수요(*need*): $n \times m$ 행렬로 각 프로세스의 각 자원에 대한 남은 수요를 나타낸다.

$$need[i, j] = max[i, j] - alloc[i, j]$$

- 표기법: X와 Y가 모두 길이가 n인 벡터일 때 $X \leq Y$일 필요충분조건은 $\forall i = 1, \ldots, n, X[i] \leq Y[i]$이다.

안전 상태 검사 알고리즘

- 사용하는 추가 테이터 구조
  - *work*와 *fin*은 길이가 각각 m과 n인 벡터이다.
- 알고리즘
  - 단계 1. $\forall i = 1, \ldots, m$, *work*$[i]$:= *avail*$[i]$로 초기화하고, $\forall i = 1, \ldots, n$ *fin*$[i]$:= false로 초기화한다.
  - 단계 2. 다음을 만족하는 i를 찾는다.
    - *fin*$[i]$ = false
    - $\forall j = 1, \ldots, m$, *need*$[i, j] \leq$ *work*$[j]$

    이런 i가 없으면 단계 4로 이동한다.
  - 단계 3. 다음을 수행하고
    - 단계 3.1. $\forall j = 1, \ldots, m$에 대해

      $$work[i] := work[i] + alloc[i, j]$$

    - 단계 3.2. *fin*$[i]$ = true

    단계 2로 이동한다.
  - 단계 4. $\forall i = 1, \ldots, n$에 대해 *fin*$[i]$ = true이면 시스템은 안전한 상태에 있는 것이다.
- 이 알고리즘은 $m \times n^2$ 연산이 필요하다.

자원 요청 알고리즘

- 사용하는 추가 테이터 구조
  - *req*$_i$: 길이가 m인 벡터로서 프로세스 P_i가 요청하는 자원의 수를 나타낸다.
- 알고리즘
  - 단계 1. $\forall j = 1, \ldots, m$에 대해 *req*$_i[j] \leq$ *need*$[i, j]$이면 단계 2로 이동하고, 그렇지 않으면 오류이다.
  - 단계 2. $\forall j = 1, \ldots, m$에 대해 *req*$_i[j] \leq$ *avail*$[i, j]$이면 단계 3으로 이동하고, 그렇지 않으면 대기한다.
  - 단계 3. 다음을 임시로 계산하고,
    - 단계 3.1. $\forall j = 1, \ldots, m$에 대해

      $$avail[i, j] := avail[i, j] - req_i[j]$$

    - 단계 3.2. $\forall j = 1, \ldots, m$에 대해

      $$alloc[i, j] := alloc[i, j] + req_i[j]$$

    - 단계 3.3. $\forall j = 1, \ldots, m$에 대해

      $$need[i, j] := need[i, j] - req_i[j]$$

    이 상태가 안전한 상태인지 안전 상태 검사 알고리즘을 이용하여 검사한다. 안전하면 할당하고, 안전하지 않으면 임시로 할당한 것을 취소한다.

은행가 알고리즘의 예

- 시스템의 구성
  - 프로세스: P_0, ..., P_4 (다섯 프로세스)
  - 자원: $A : 10$, $B : 5$, $C : 7$ (세 가지 유형과 각 유형의 인스턴스 수)
- 시간 t_0 때 시스템의 상태가 다음과 같다고 하자.

	alloc			*max*			*avail*		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

그러면 *need*$[i, j]$는 *max*$[i, j]$-*avail*$[j]$이므로 다음과 같다.

	need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

- 먼저 이 시스템이 안전한 상태인지 검사하여 보자.
  - *work* $= \{3, 3, 2\}$이므로 P_1과 P_3은 단계 2를 만족한다.
  - P_1을 먼저 수행하면 *work* $= \{5, 3, 2\}$가 된다.

- P_3 또는 P_4을 수행할 수 있다.
- P_4을 수행하면 *work* $= \{5, 3, 4\}$가 된다.
- P_3 밖에 수행할 수 없다.
- P_3을 수행하면 *work* $= \{7, 4, 5\}$가 된다.
- P_0 또는 P_1을 수행할 수 있다.
- P_0을 수행하면 *work* $= \{7, 5, 5\}$가 된다.
- P_1을 수행할 수 있다.
- 즉 $< P_1, P_4, P_3, P_0, P_1 >$은 안전한 순서이므로 이 시스템은 현재 안전한 상태이다.

- 이 때 P_1이 $req_i = \{1, 0, 2\}$을 요청하였다고 하자. 이 요청을 허용할 수 있는지 검사하여 보자. 이 요청은 *need*$[1]$보다 적고, *avail*보다 적으므로 임시로 할당하면 다음과 같다.

	alloc			*need*			*avail*		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

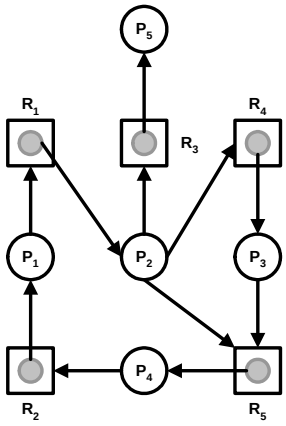
  - *work* $= \{2, 3, 0\}$이므로 P_1밖에 수행할 수 없다.
  - P_1을 수행하면 *work* $= \{5, 3, 2\}$가 된다.
  - 이 이후는 기존과 같은 순서로 실행이 가능하다. 따라서 이 할당은 허용된다.

8.6 교착상태 탐지

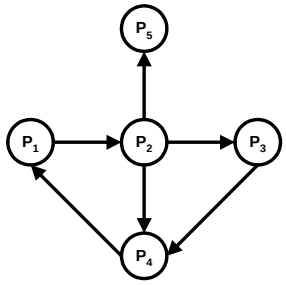
- 교착상태 탐지 방법이란 교착상태를 허용하고 주기적으로 교착상태를 발견하여 시스템을 복구하는 방법이다.
- 이 방법을 사용하기 위해서는 교착상태를 발견하기 위해 필요한 정보를 시스템에서 유지해야 한다. 뿐만 아니라 교착상태로부터 회복하는 과정에서 본질적으로 손실이 발생한다.

8.6.1 각 자원의 유형마다 단일 인스턴스로 구성된 경우

- 각 자원의 유형마다 단일 인스턴스로만 구성된 경우에는 자원 할당 그래프를 이용하여 교착상태를 발견할 수 있다.
- 자원 할당 그래프를 먼저 **대기 그래프(wait-for graph)**로 변경한다. 변경하는 방법은 다음과 같다.
  - $P_i \rightarrow P_j$의 의미: P_i는 P_j가 점유하고 있는 자원을 기다린다.
  - 자원 할당 그래프에 있는 프로세스만을 이용하여 대기 그래프를 작성한다.



(a) 자원 할당 그래프



(b) 대기 그래프

<그림 8.5> 자원 할당 그래프와 그것에 대응되는 대기 그래프

  - 자원 할당 그래프에 요청선 $P_i \rightarrow R_i$와 할당선 $R_i \rightarrow P_j$가 존재하면 대응되는 대기 그래프에 $P_i \rightarrow P_j$를 추가한다.
- 교착상태은 대기 그래프에 주기가 있는지 검사하여 발견한다.

8.6.2 각 자원의 유형이 여러 인스턴스로 구성된 경우

- 은행가 알고리즘과 유사한 다음과 같은 데이터 구조를 이용한다.
  - 사용 가능(*avail*): 길이 m인 벡터로 각 자원의 유형마다 사용 가능한 자원의 수를 나타낸다.
  - 할당(*alloc*): $n \times m$ 행렬로 각 프로세스에게 할당되어 있는 각 자원이 인스턴스의 수를 나타낸다.
  - *req*: $n \times m$ 행렬로 각 프로세스가 각 자원의 유형마다 요청하고 있는 자원의 수를 나타낸다.
  - *work*와 *fin*은 길이가 각각 m과 n인 벡터이다.
- 탐지 알고리즘
  - 단계 1. $\forall i = 1, \ldots, m$에 대해 *work*$[i]$:= *avail*$[i]$로 초기화하고, $\forall i = 1, \ldots, n$에 대해 *alloc*$[i]$가 0이 아니면(즉, $\forall j = 1, \ldots, m$에

대해 $alloc[i,j] \neq 0$) $fin[i] :=$ false로 초기화하고, 그렇지 않으면 $fin[i] :=$ true로 초기화한다.

- 단계 2. 다음을 만족하는 i를 찾는다.
  - $fin[i] =$ false
  - $\forall j = 1, \ldots, m$에 대해

$$req[i,j] \leq work[j]$$

  이런 i가 없으면 단계 4로 이동한다.
- 단계 3. 다음을 수행하고
  - 단계 3.1. $\forall j = 1, \ldots, m$,

$$work[i] := work[i] + alloc[i,j]$$

  - 단계 3.2. $fin[i] =$ true

  단계 2로 이동한다.
- 단계 4. $fin[i] =$ false인 $i = 1, \ldots, n$이 존재하면 시스템은 교착상태에 있다.

교착상태 탐지 알고리즘의 예

- 시스템의 구성
  - 프로세스: P_0, ..., P_4 (다섯 프로세스)
  - 자원: $A : 7$, $B : 2$, $C : 6$ (세 가지 유형과 각 유형의 인스턴스 수)
- 시간 t_0 때 시스템의 상태가 다음과 같다고 하자.

	alloc			*req*			*avail*		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

- 이 시스템이 교착상태인지 검사하여 보자.
  - $work = \{0,0,0\}$이고, 모든 i에 대해 $fin[i] =$ false이다.
  - 따라서 P_0과 P_2가 단계 2를 만족한다.
  - P_2를 먼저 수행하면 $work = \{3,0,3\}$이 된다.
  - P_0, P_1, P_3, P_4을 수행할 수 있다.
  - P_3을 수행하면 $work = \{5,1,4\}$가 된다.
  - P_0, P_1, P_4을 수행할 수 있다.
  - P_4을 수행하면 $work = \{5,1,6\}$이 된다.
  - P_0 또는 P_1을 수행할 수 있다.
  - P_1을 수행하면 $work = \{7,1,6\}$이 된다.
  - P_0을 수행할 수 있다.
  - 즉 $< P_2, P_3, P_4, P_1, P_0 >$은 안전한 순서이므로 이 시스템은 교착상태가 아니다.
- 이 때 P_2가 자원 유형 C을 요청하여 한 인스턴스를 할당받아 다음과 같이 상태가 변경되었다고 하자.

	alloc			*req*			*avail*		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	1			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

- 이 시스템이 교착상태인지 검사하여 보자.
  - $work = \{0,0,0\}$이고, 모든 i에 대해 $fin[i] =$ false이다.
  - 따라서 P_0만 단계 2를 만족한다.
  - P_0를 먼저 수행하면 $work = \{0,1,0\}$이 된다.
  - 이 이후에는 실행할 수 있는 프로세스가 없다.
  - 따라서 P_1, P_2, P_3, P_4은 교착상태에 있다.

8.6.3 탐지 알고리즘의 사용

- 탐지 알고리즘의 사용 빈도를 결정하는 요소
  - 쟁점 1. 교착상태가 발생하는 빈도
    교착상태가 자주 발생하면 탐지 알고리즘을 자주 실행해야 한다.
  - 쟁점 2. 교착상태가 발생하였을 때 영향을 받는 프로세스의 수
    교착상태를 오래동안 방치하면 교착상태에 포함되는 프로세스의 수는 증가한다.
- 교착상태는 프로세스가 바로 할당받을 수 없는 자원을 요청하였을 때 발생한다.
- 극단적인 경우는 바로 할당될 수 없는 자원을 요청할 때마다 탐지 알고리즘을 수행할 수 있다. 이렇게 하면 교착상태에 포함되는 모든 프로세스를 알 수 있을 뿐만 아니라 교착상태를 일으킨 프로세스를 알 수 있다. (말이 모순적임) 그러나 이 방법은 비용 때문에 현실성이 없으며 예방에 가까운 방법이다.
- 자원의 유형이 많으면 하나의 요청이 여러 주기를 만들 수 있다.
- 일반적으로 탐지 알고리즘 실행 빈도를 한 시간에 한번 정도로 줄인다. 또는 CPU 사용 효율이 40% 이하로 떨어진 경우에 실행한다.

8.6.4 교착상태로부터의 회복

- 교착상태를 발견하면 시스템 운영자에게 통보하여 수동으로 처리하는 방법이 있고, 시스템이 자동으로 복구하는 방법이 있다.
- 교착상태로부터 회복하는 두 가지 방법

  - 순환 대기하는 프로세스를 종료
  - 교착상태에 포함되어 있는 프로세스에서 하나 이상의 자원을 강제로 해제하는 방법

8.6.5 프로세스 종료

- 두 가지 방법
  - 방법 1. 교착상태에 포함된 모든 프로세스를 종료: 교착상태는 바로 해결되지만 비용이 많이 소요된다.
  - 방법 2. 순환 대기하는 프로세스 중 하나를 종료: 비용은 적지만 바로 교착상태가 해결되지 않을 수 있어 프로세스를 추가로 종료해야 하는 경우가 발생할 수 있다.
- 프로세스 종료의 문제점
  - 작업 중간에 강제 종료하면 일광성 문제가 발생할 수 있다. 예) 파일 갱신 중에 종료
  - 작업 중간에 강제 종료하면 하드웨어를 초기화해 주어야 하는 경우가 발생할 수 있다. 예) 프린터
- 위에서 방법 2를 사용하면 어떤 프로세스를 종료할지 결정해야 한다. 이 결정에 영향을 주는 요소는 다음과 같다.
  - 프로세스의 우선순위
  - 프로세스의 실행시간, 남은 시간
  - 프로세스가 점유하고 있는 자원의 수와 유형(선점하기 쉬운 자원을 가지고 있는 프로세스를 종료하는 것이 유리하다.)
  - 프로세스가 필요하는 자원의 수
  - 교착상태를 해결하기 위해 종료해야 하는 프로세스의 수
  - 프로세스의 종류(시스템, 상호작용 등)

8.6.6 자원의 강제 해제

- 자원의 강제 해제 방법을 사용할 때 고려되는 요소
  - 희생자 선택: 어떤 자원을 어떤 프로세스로부터 해제할지 결정해야 한다. 비용을 계산할 때 고려되는 요소는 프로세스를 종료할 때와 유사하다.
  - 복귀(rollback): 어떤 프로세스로부터 자원을 강제 해제하면 그 프로세스는 다시 정상 수행될 수 있도록 이 프로세스를 안전한 상태로 복귀시켜야 한다. 가장 단순한 방법은 처음부터 다시 실행하는 것이다. 반대로 가장 이상적인 것은 필요한 만큼만 되돌리는 것이다. 하지만 이 경우에는 시스템이 많은 정보를 유지해야 한다.
  - 굶주림: 같은 프로세스로부터 계속 자원을 해제하면 이 프로세스는 영구적으로 실행되지 못할 수 있다. 이런 현상이 발생하지 않도록 해야 한다. 보통 희생자로 선택될 수 있는 회수를 제한한다.