**[운영체제]**

<컴퓨터 하드웨어 구성>

1. 프로세서

- 시스템 버스 : 프로세서, 메모리, 주변장치를 연결

- 레지스터 : CPU안에 존재

> 사용자 가시 레지스터

>> 데이터 레지스터 DR : 함수연산에 필요한 데이터를 저장. 산술연산, 논리연산에 사용 연산 결과로 플래그 값 저장

>> 주소 레지스터 AR : 주소, 유효주소 계산에 필요한 주소의 일부분 저장.

>>> 기준 주소 레지스터 : 프로그램 실행 시 사용하는 기준 주소 값을 저장. 기준 주소는 하나의 프로그램이나 일부 등 서로 관련있는 정보를 저장하며, 연속된 저장 공간을 지정하는데 참조가능한 주소. 따라서 페이지나 세그먼트처럼 블록화된 정보에 접근하는데 사용됨.

>>> 인덱스 레지스터 : 유효 주소를 계산하는데 사용하는 주소 정보 저장

>>> 스택 포인터 레지스터: 메모리에 프로세서 스택을 구현하는데 사용. 많은 프로세서와 주소 레지스터를 데이터 스택 포인터와 큐포인터로 사용

> 사용자 불가시 레지스터 : 사용자가 정보를 변경할 수 없는 레지스터

>> 프로그램 카운터 PC : 다음에 실행할 명령어의 주소를 보관하는 레지스터

>> 명령어 레지스터 IR: 현재 실행 중인 명령어를 저장하는 레지스터

>> 누산기 ACC: 데이터를 일시적으로 저장하는 레지스터

>> 메모리 주소 레지스터 MAR : 프로세서가 참조하려는 데이터 주소를 명시하여 메모리에 접근하는 버퍼 레지스터

>> 메모리 버퍼 레지스터 MBR (메모리 데이터 레지스터) : 프로세서가 메모리에서 읽거나 저장할 데이터 자체를 보관하는 버퍼 레지스터

2. 메모리

- 메모리 속도와 가격 : 보조기억장치 < 메인 메모리 < 캐시 < 레지스터 (용량은 반대)

- 메인 메모리 : CPU 외부에 있으며 프로세서에서 즉각적으로 수행할 프로그램, 데이터를 저장하거나 프로세서에서 처리한 결과를 메인 메모리에 저장한다.

> 저장 밀도가 높고 싼 DRAM을 주로 사용한다.

> 다수의 셀로 구성되며, 각 셀은 비트로 구성되고, 셀이 K 비트이면 2^K 값을 저장할 수 있다.

> 물리적 주소 : 셀은 주소로 참조하며, 주소가 n비트이면 주소 범위는 0 ~ 2^n – 1 이다.

- 논리적 주소 : 컴파일러가 프로그램을 기계어로 변환할 때 변수와 명령어에 할당하는 주소

> 매핑 : 컴파일로 논리적 주소를 물리적 주소로 변환하는 것

3. 캐시

프로세서 내부나 외부에 존재하며 처리속도가 빠른 CPU와 상대적으로 느린 메인 메모리의 속도차이를 보완하는 고속 버퍼이다.

<컴퓨터 시스템의 동작>

2. 명령어 실행

- 명령어 인출 > 명령어 해석, PC 변경 > 피연산자 인출 > 명령어 실행 > 결과 저장

<프로세스와 스레드>

1. 프로세스의 개념

- 스택 : 데이터를 일시적으로 저장하는 영역.

- 힙 : 코드 영역과 별도로 유지되는 자유 영역. 동적으로 메모리 할당하는데 사용.

2. 프로세스 상태변화

- 프로세스 생성

> 준비 : 비실행 상태

> 실행중인 프로세스 종료 나 인터럽트 발생시 디스패치

> (이벤트, 입력신호에 의한 대기상태 > 이벤트, 입출력 완료시) 준비상태 or 할당 시간 종료시 준비 상태

> 실행 > 종료> time out : 독점 방지

- 프로세스 제어 블록 PCB == TCB(Task Control Block): 프로세스 제어 시 필요한 프로세스 상태 정보 저장

> 프로세스 생성 시 메모리에 PCB 생성, 실행 종료시 삭제.

>> 프로세스 식별자 : 각 프로세스 고유 식별자(숫자, index 항목)

>> 프로세스 상태 : 생성, 준비, 실행, 대기, 중단

>> 프로그램 카운터 : 다음 명령의 주소 표시

>> 레지스터 저장 영역 : 누산기, IR, 스택 포인터, 범용레지스터, 조건 코드등의 정보인터럽트 발생 시 PC와 함께 저장하여 재 실행 시 원래대로 복귀할 수 있게 함

>> 프로세서 스케줄링 정보 : 우선순위, 스케줄링 큐의 포인터, 기타 스케줄 매개변수

>> 계정 정보 : 프로세서 사용시간, 실제 사용시간, 사용 상한 시간, 계정 번호 등등

>> 입출력 상태 정보, 메모리 관리 정보

- 프로세스 문맥 교환(PCS Switching) : 실행중인 프로세스가 제어를 다른 프로세스에 넘겨 실행 상태가 되도록 하는 것. 문맥 교환시 프로세서 레지스터에 있던 내용을 나중에 사용할 수 있도록 저장함

<스레드의 개념과 상태변화>

1. 스레드의 개념 : 프로세스의 제어 특성만 분리한 실행 단위. 프로세스는 한 개 이상의 스레드로 나눌 수 있다. 한 개의 프로세스에 대하여 병렬처리를 위하여 사용한다.

- 경량 프로세스 LWP Light Weighted process : 프로세서의 속성 중 일부가 들어가 있는 것

- 중량 프로세스 HWP : 스레드 하나에 프로세스가 하나인 경우.

- 스레드는 PC, SP, 문맥정보, 지역데이터, 스택을 독립적으로 가지고, 코드, 전역데이터, 힙을 다른 스레드와 공유한다.

- 프로세스를 생성하는 것보다 스레드를 생성하여 문맥 교환하는 것이 오버헤드가 적어서 경제성이 좋다.

4. 스레드 상태 변화

- 프로세스의 상태 변화와 거의 동일하다. 단지 실행단계에서 스레드 하나가 해당 프로세스를 점유하여 실행된다.

- 스레드 하나가 대기 상태로 가면 프로세스 전체가 대기 상태로 가는 게 아니라 다음 스레드가 프로세서를 점유하여 실행된다.

- 스레드 제어 블록 TCB : 스레드의 정보를 저장한다. 프로세스는 여러개의 스레드를 가질 수 있으므로 TCB는 리스트 형태이다.

<스레드의 구현>

- 사용자 수준 스레드 : 스레드 라이브러리를 이용하여 작동, 다대일 매핑 (커널이 1, 스레드라이브러리가 다)

- 커널 수준 스레드 : 커널(운영체제)에서 지원하는 형태, 일대일 매핑

- 혼합형 스레드 : 위 둘을 혼합한 형태, 다대다 매핑

1. 사용자 수준 스레드

- 스레드와 관련된 모든 행위를 사용자 영역에서 하므로 커널이 스레드의 존재를 모름.

- 스레드 라이브러리 : 스레드 생성, 종료, 스레드간 메시지 전달, 스레드의 스케줄링과 문맥 정보 보관

- 커널은 스레드가 아닌 프로세스를 한 단위로 인식하고 프로세서를 할당, 즉 프로세스 테이블 유지

- 다수의 사용자 수준 스레드가 하나의 커널 수준 스레드에 매핑되므로 다대일 스레드 매핑

- 커널에서 스레드를 지원할 필요가 없음

- 장점

> 이식성 높음 : 커널에 독립적으로 스케줄링 가능해 모든 OS에 적용 가능

> 오버헤드 적음 : 스케줄링, 동기화를 위한 커널 호출이 없으므로 커널 영역 전환이 없고 오버헤드가 줄어들음

> 유연한 스케줄링 : 스레드 라이브러리에서 스케줄링을 제어하므로 응용프로그램에 맞게 스케줄링 가능

- 단점

> 시스템의 동시성을 지원하지 않음 : 프로세스 단위로 프로세서를 할당하여 다중 처리 환경을 갖춰도 스레드 단위로 다중 처리를 하지 못함. 즉 스레드 한 개가 대기상태가 되면 해당 프로세스 내의 어떤 스레드도 실행 못함

> 확장에 제약이 따름 : 커널이 한 프로세스 내의 여러 스레드에 프로세서를 동시에 할당할 수 없으므로 다중 처리 시스템에서 규모 확장이 어려움

> 스레드 간 보호 불가능 : 스레드 간 보호에 커널의 보호 방법 사용 불가능. 스레드 라이브러리에서 스레드 간 보호를 제공해야 프로세스 수준에서 보호가 가능

2. 커널 수준 스레드

- 커널이 스레드 관련 모든 작업을 관리. TCB, PCB 유지

- 한 프로세스에서 다수의 스레드가 프로세서를 할당 받아 병행으로 수행 하고, 스레드 한 개가 대기상태가 되면 해당 프로세스 내 다른 스레드로 교환이 가능함. 단 커널이 개입하므로 커널영역으로의 전환이 필요. 오버헤드 발생

- 사용자 수준 스레드와 커널 수준 스레드가 일대일로 매핑됨

- 커널이 직접 스케줄링하고 실행하므로 사용자 수준 스레드의 커널 지원이 부족한 문제를 해결 가능. 단, 전체 프로세스와 스레드 정보를 유지해야 하므로 오버헤드가 커짐

3. 혼합형 스레드

- 좀 복잡하다

- 사용자 수준 스레드는 경량 프로세스에 다대다로 매핑되고, 경량 프로세스는 커널 수준 스레드와 일대일로 매핑된다. 즉, 다수의 사용자 수준 스레드에 다수의 커널 스레드가 다대다로 매핑된다.

- 커널 수준 스레드는 디스패치하고 스케줄링하여 프로세서에서 실행한다. 경량 프로세스는 시스템 호출로 생성해서 커널 영역의 프로세스 문맥 안에서 실행하고, 커널로 독립적으로 스케줄링하여 다중 처리에서는 병렬로 실행한다.

- 자원과 입출력 대기를 위하여 경량 프로세스 단위로 대기하므로 프로세스는 입출력을 완료할 때까지 대기할 필요가 없다. 어떤 경량 프로세스가 입출력 완료를 기다리더라도 해당 프로세스 내의 다른 경량 프로세스를 실행할 수 있기 때문이다.

-**스레드 풀링** : 시스템이 관리하는 스레드 풀을 응용 프로그램에 제공하여 미리 생성한 스레드를 재사용 하도록 하여 스레드 생성 시간을 줄여서 시스템의 부담을 덜어줌.

또한 동시에 생성할 수 있는 스레드 수를 제한하여 시스템의 자원 소비를 줄여서 응용 프로그램의 전체 성능을 일정수준으로 유지함

**<병행 프로세스>**

운영체제가 프로세서를 빠르게 전환하여 프로세서 시간을 나눠서 마치 프로세스 여러 개를 동시에 실행하는 것처럼 보이게 하는 것

-독립 프로세스 : 다른 프로세스에 영향을 주고받지 않으면서 독립적으로 실행하는 프로세스. 주어진 초기 값에 대하여 항상 같은 결과를 보여줌

> 단일 프로그래밍 : CPU를 사용하던 프로세스 완료 후 다음 프로세스를 실행한다

**> 다중 프로그래밍 : 프로세스 여러 개가 하나의 CPU를 공유한다.**

**> 다중 처리 : 프로세서를 2개이상 사용하여 동시에 프로그램 여러 개를 병렬로 실행한다.**

**-협력 프로세스 : 다른 프로세스에 영향을 주고 받으며 특정 기능을 수행하는 비동기적 프로세스.**

> 컴퓨터 자원 효율성 증대, 계산 속도 향상, 모듈적 구성 강화, 여러 작업 동시 처리 가능

**> IO, 메모리, CPU, 클록 등의 자원의 사용에 대해서 프로세스들 간에 충돌 발생 가능. 그래서 다음과 같은 상호작용이 필요**

>> 1. 프로세스는 서로 인식하지 못하는 경쟁관계 유지. 이때 **상호배제가 필요**

>> 2. 프로세스는 입출력 버스를 비롯한 개체를 공유하는 단계에서 간접적으로 서로의 관계를 인식. 개체 공유에 따른 협력이 필요함

>> 3. 서로를 인식하고 프로세스끼리 통신할 수 있는 기본 함수가 존재

> 병행 시스템은 다중 처리 시스템, 분산 처리 시스템 등에서 중요 개념으로 사용된다.

>> 병행성은 시스템 신뢰도를 높이고 처리 속도를 개선하기 위해 다음 문제들을 해결해야한다.

**>>>** **공유 자원을 “상호 배타적”으로 사용해야 한다.** 즉, 한순간에 하나의 프로세스만이 자원에 접근해야 한다.

**>>> 병행 프로세스 간에는 협력이나 동기화가 되어야 한다. 상호배제도 동기화의 한 형태이다.**

>>> 두 프로세스 사이에서 데이터를 교환할 수 있도록 통신 되어야 한다.

>>> **프로세스는 다른 프로세스와 관계 없이 항상 일정한 실행 결과를 보장하도록 결정성을 확보해야 한다.**

>>> **교착 상태를 해결하고 병렬 처리 능력을 극대화 해야 한다.**

>>> 실행 검증 문제를 해결해야 한다.

>>> **상호배제를 보장해야한다.**

**<상호배제(Mutex)와 동기화>**

1. 상호배제의 개념

* 병행 프로세스에서 프로세스 하나가 공유 자원을 사용할 때 다른 프로세스 들이 해당 자원에 접근 할 수 없도록 하는 방법
* **동기화 : 순차적으로 재사용 가능한 자원을 공유하려고 상호작용하는 프로세스 사이에서 발생**
* 공유자원을 동시에 사용하지 못하게 실행을 제어하는 방법
* **상호 배제는 프로세스가 수정할 수 있는 공유 데이터에 접근할 때만 적용한다.**

**<임계구역>**  
- 다중 프로그래밍 OS에서 공유되는 데이터 또는 자원에 대해서 **한번에 하나의 프로세스만이 접근 가능한 공유 자원을 말한다.  
> 한 개의 프로세스 독점할 수 없으며, 임계 영역에서 수행중인 프로세스는 인터럽트가 불가능하다.**

* 1. **상호배제를 보장해야 한다.**
* 2. 진행 : 임계영역이 비어있는 상태에서 여러 프로세스가 임계영역에 접근하려고 하면 **어떤 프로세스가 들어갈지 적절히 결정**해야 한다.
* 3. 한정 대기 : 임계 영역을 위해 무한정 기다리는 것을 방지하기 위해, **임계 영역에 한번 들어갔던 프로세스는 다음에 임계 영역에 다시 들어갈 때 제한을 둔다.**

<상호배제 방법들>

* 소프트웨어 기준
* 두개 프로세스 : 데커, 피터슨
* 여러 프로세스 : Lamport 빵집 알고리즘
* 하드웨어 기준
* Test and Set , Swap 명령어 기법

1. 데커 알고리즘

* 임계 영역 진입시 **flag[0]=True 를 설정한 후** 차례를 기다린다. Flag[1]이 true이면 임계영역에 **누가 있는 상태**이다. **그럼 flag[0]=False 로 바꾸고 기다린다.**
* ex) 나 들어갈꺼야 하고 깃발 꽂았는 데, 이미 다른 깃발이 꽂혀 있으면, 그 깃발뽑힐때까지 기다리는 것이다. == **선 선언 후 확인.**

1. Test and Set (TAS) 명령어

* **원자적 연산 : 중단 없이 실행하고 중간에 다른 사람이 수정할 수 없는 최소단위 연산이다.**
* Lock을 false로 초기화 한 후 **프로세스가 임계영역에 들어가면 lock 이 true가 된다.** 그후 임계 영역에서의 **작업을 마치면 lock이 false가 되어 임계영역이 열린다.**
* 여러 프로세스를 위해서는 **waiting[i] 배열을 사용하여 true이면 임계영역에 접근한다는 것을 표기한다.**(즉, 나 기다리고있음! 이라고 표시하는 거임) **그리고 key를 true로 하여 임계영역에 접근하고 lock 을 true로 하여 임계 영역을 닫는다**.(즉, key는 이번에 내가 임계영역에 들어가 있음을 나타내고, lock은 임계영역에 누가 있으니 문 닫는다? 라는 의미이다.) **Lock 이 false인 경우 key 값이 false가 되어 임계영역에 접근할 수 있다.**
* 그후 waiting 배열에서 차례가 높은 프로세스를 **시작한후 waiting 값이 true인 첫번째 프로세스가 다음에 임계영역에 진입할 프로세스가 된다**. **이때 waiting 값이 true인 프로세스가 없으면 lock을 false로 한다.**
* 단점
* >> 바쁜 대기 발생 : 프로세서 시간 소모 크고 **대기프로세서는 비생산적이며 자원이 소모되는 대기 루프에 남는다**
* >> 기아 상태 발생 : 프로세스가 원하는 자원을 얻지 못해 계속 기다리는 현상.
* >> 교착 상태 발생 : 플래그는 우선순위가 낮은 프로세스가 재설정 가능하지만, 우선순위가 높은 프로세스가 선점한다. 그러므로 우선순위 낮은 프레스는 lock을 가지고, 우선순위가 높은 프로세스가 이것을 얻으려 시도할 때 높은 우선순위 프로세스는 무한정 바쁜 대기가 될것이다.
* 위에 두개는 밑에서 다시 보자.

1. **세마포**

* 반복적인 임계영역 조사에 의하여 프로세서 사이클이 낭비되는 것을 다익스트라가 동기화 도구인 세마포를 제안하여 해결했다.
* **동기화** 기법 : 두개 이상의 프로세스를 동시에 처리할 수 없으므로 **각 프로세스에 대한 처리 순서를 결정하는 것으로, 상호배제의 한 형태이다.**
* 동기화 구현방법으로는 **세마포와 모니터가 있다.**
* 세마포란 각 프로세스에 제어 신호를 전달하여 순서대로 작업을 수행하도록 하는 기법이다.
* S(사용가능 자원 개수)는 P와 V 연산으로만 접근 가능한 세마포어 변수로, **공유 자원의 개수를 나타내며 0과 1 혹은 0과 양의값을 가질 수 있다. (0과 1만 가진 것을 이진 세마포라고 하며 이거는 그냥 임계구역 나누는 거랑 똑같다.)**
* **P연산은 임계영역에 진입하기 위하여 프로세스를 대기하게 하며 S값을 검사하여 양수이면 1을 감소시키는 과정이고, V 연산은 대기중인 프로세스를 깨우려 신호를 보내는 동작이며 S를 1만큼 증가시키는 과정이다. 🡸 여기가 중요한데, 즉 P연산은 “나 이제 임계구역 들어간다?” 라고 말하고 사용가능 자원(S)S을 한 개 줄이는 것이다. 그리고 V연산은 “나 임계구역 다 썼음. 다음 쓸사람 쓰셈.” 하고 사용가능 자원(S)을 한 개 증가시키는 것이다.**
* 프로세스는 **임계 영역에 진입하려고 wait(S)**를 호출하며 이는 다른 프로세스가 대기 조건을 통과하는 것을 방지할 수 있다.
* 해당 프로세스가 **임계 영역을 종료하면 다음 프로세스가 임계 영역에 진입할 수 있도록 signal(S)를 호출**한다.
* **이진 세마포 (쉽게 자원이 1개인 상태라고 생각하면 된다.)**
* **P(S) : S를 검사하여 양수이면 S를 0으로 재설정 한 후 진행**(임계영역 입장)하고**, 아니면 S를 준비큐로 되돌린다.**
* **V(S) : S를 1로 설정하고 준비 큐에 있는 프로세스를 시작한다.**

1. **모니터(임계구역에 무조건 1개의 프로세스만 들어간다.)**

* 동기화 구현을 위한 특수 프로그래밍 기법으로 **특정 공유 자원을 프로세스에게 할당하는데 필요한 데이터와 데이터를 처리하는 프로시저로 구성된다.**
* **자료 추상화와 정보 은폐 개념을 기초**로 하며 공유자원을 할당하기 위한 병행성 구조로 이루어져 있다.
* **모니터 내의 공유자원을 사용하려면 프로세스는 반드시 모니터 진입부를 호출해야 한다.**
* **> 즉 임계구역사용()이라는 메소드를 한명만 사용할 수 있다는 것이다. 이 함수를 Synchronized()라고한다.**
* **외부의 프로시저는 직접 액세스 할 수 없다.**
* 모니터의 **경계에서 상호 배제가 시행**된다.
* 모니터에는 **한 순간에 하나의 프로세스만 진입하여 자원을 사용할 수 있다.** 즉 상호배제가 보장된다.
* 모니터에는 Wait과 Signal 연산이 사용된다.
* 이미 **모니터에서 실행 중인 다른 프로세스가 있으면 다른 호출 프로세스는 차단**한다 즉, 진입권한을 **차단해 모니터의 외부 모니터 준비큐에서 대**기하고 **모니터가 비면 준비 큐의 프로세스 중 하나를 차단 해제**하고 **모니터 프로시저를 호출할 수 있는 권한을 부여한다**

<세마포와 뮤텍스(상호배제)와 모니터의 차이>

- 뮤텍스는 뮤텍스를 소유하고 있는(lock) 스레드가 lock을 해제할 수 있다. 즉 열쇠를 가진 사람만(자물쇠를 가진 사람)이 자물쇠를 해제할 수 있다.

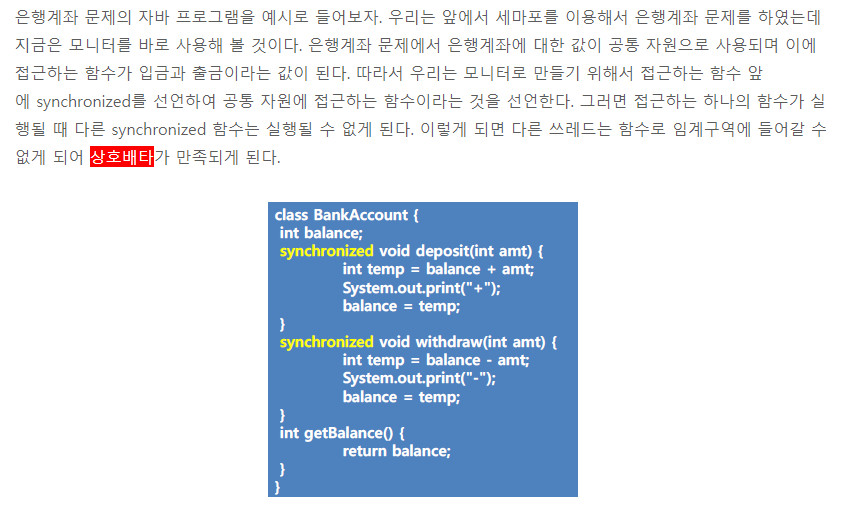
- 세마포는 그냥 뮤텍스에서 키가 다수인 것이다.

- 자바의 모든 객체는 Object 클래스를 상속 받는다. object 클래스는 wait(), notifyAll(), notify()메소드를 가지고 있고, 이것이 Condition Variables의 역할을 한다. 즉 **모든 자바 객체는 모니터를 가지고 있다.**

> 모니터로 상호배제를 해결하기 위하여 Synchronized 구현체를 사용한다. 쓰레드 A가 임계구역 Q의 Lock을 가지고 임계구역사용 메소드를 실행 중이면, 쓰레드 B는 그 임계구역 Q의 Lock을 얻어야 Q에 접근할 수 있다.

>> 이때 A가 Q의 lock을 반환해야지만 B가 Q의 Lock을 얻어 Q에 접근할 수 있다.

>> 모니터는 공유자원에 접근할 수 있는 키의 획득과 해제를 관리한다. 하지만 세마포는 직접 키해제와 공유자원 처리를 해주어야한다.



<교착 상태의 개념과 발생 원인>

1. 교착 상태의 개념 deadlock

* **교착 상태 deadlock : 다중 프로그래밍 시스템에서 프로세스가 결코 일어나지 않을 사건을 기다리는 상태**
* **둘 이상의 프로세스가 다른 프로세스가 점유하고 있는 자원을 서로 기다릴 때 발생**
* 운영체제가 해결 못하면 사용자가 외부 간섭으로 해결해야함
* 프로세스가 자원을 사용하는 순서
* 자원 요청 : 필요 자원을 요청한다. 다른 프로세스가 사용 중이면 기다려야 한다.
* 자원 사용 : 자원을 획득하여 사용한다.
* 자원 해제 : 자원 사용을 마친 후 해당 자원을 해제 해준다.
* **교착 상태** **발생 조건**
* 상호 배제 : 자원을 최소 하나 이상 비공유해야 한다**. 즉, 한번에 하나의 프로세스만이 해당 자원을 사용할 수 있어야 한다.**
* 점유와 대기 : 자원을 **최소 하나를 보유한 채로, 다른 프로세스가 할당 받은 자원을 기다리는 프로세스가 있어야 한다.**
* 비선점 : 자원은 선점할 수 없다. **즉 자원을 강제로 뺏을 수 없다.**
* 순환(환형)대기 : 프로세스들이 **서로 가진 자원을 다른 프로세스들이 얻으려고 기다리는 상태를 말한다.**
* **선점 자원 :** 부작용 없이 소유한 프로세스 에서 **빼앗아 선점할 수 있는 자원**. Ex)CPU
* **비선점 자원** : 하나의 프로세스에서 빼앗아 선점할 수 없고, 부작용 없이 다른 프로세스에 할당할 수 없는 자원. Ex) **임계영역**
* **보통 교착상태는 비선점 자원이 발생시킨다.**

<교착 상태의 해결 방법>

1. **교착 상태 예방**

* 자원의 상호배제 조건 방지

> 근데 상호배제를 해제해 버리면 공유되면 안되는 자원이 공유되버려서 안될 것 같다.

* **점유와 대기 조건 방지**
* 점유와 대기가 발생하지 않으려면, **프로세스가 작업을 수행하기 전에 필요한 자원을 모두 요청하고 획득해야 한다. 즉, 내가 필요한 자원을 모두 알고 있고, 모두 가진 상태에서만 실행해야 한다.**
* **프로세스가 자원을 전혀 가지고 있지 않을 때만 자원 요청이 가능하고,** 한번에 하나의 프로세스에 모든 자원을 할당하여 실행 후 다음 프로세스로 넘어가는 것이다.
* 무한 대기 상태가 발생할 수 있다 : **자주 사용하는 자원이 다른 프로세스에 할당되어 있는 프로세스는 무한정 기다려야 하는 경우가 발생한다.**
* 비선점 조건 방지
* 이미 할당된 자원에 선점권이 없어야 한다.
* **어떤 자원을 요청할 때 거부되면, 해당 프로세스는 자신이 가진 자원을 모두 버리고 대기해야한다**. 그에 대한 대안으로, **요청전에 해당 자원을 사용할 수 있는지 확인한다. 만약 해당 자원을 대기 프로세스가 가지고있으면 해당 자원을 해제하여 가져온다.**

**>> 즉, 내가 필요한 자원을 모두 확인했는데, 다른 프로세스가 가지고 있어, 근데 그 프로세들이 대기상태야. 그럼 뺏어버리는 거지.**

* 또는 높은 우선순위 프로세스가 낮은 우선순위 프로세스가 점유한 자원을 선점하여 해결한다.
* 순환 대기 조건 방지
* **모든 자원에 일련의 순서를 부여하고 각 프로세스가 오름차순으로만 자원을 요청할 수 있게 한다.** 단 이방법은, **예상 순서와 다르게 자원을 요청하는 작업에 대하여 오랫동안 자원을 할당 받은 상태로 있어야 하므로 상당한 자원 낭비를 초래한다.**

>> 예방 방법들은 장치의 효율성과 시스템 처리량을 떨어뜨린다.

1. **교착 상태 회피**

* 프로세스의 시작 중단
* 자원 할당 거부 (은행가 알고리즘)
* 자원의 **할당 허용 여부를 결정하기 전에 미리 결정된 모든 자원의 최대 가능한 할당량을 시뮬레이션하여 안전 여부를 검사한다**. 그 후 대기 중인 모든 활동의 교착 상태 가능성을 조사하여 안정 상태 여부를 검사, 확인한다.
* **각 프로세스가 요청하는 자원 종류의 최대 수를 알아야한다.**
* E,J,Dijkstra가 제안한 방법으로, 은행에서 모든 고객의 요구가 충족되도록 현금을 할당하는 데서 유래한 기법이다.
* **프로세스가 자원을 요구할 때 시스템은 자원을 할당한 후에도 안정 상태로 남아있게 되는지를 사전에 검사하여 교착 상태를 회피하는 기법**
* **안정 상태에 있으면 자원을 할당하고, 그렇지 않으면 다른 프로세스들이 자원을 해지할 때까지 대기함**
* **여기서 안정이란, 각 프로세스에 최대치까지 자원을 할당할 수 있고 교착상태를 예방할 수 있는 상태를 말한다.**
* 해당 프로세스가 요구하는 자원을 다른 프로세스가 점유 중이면 기다린다.

1. 교착 상태 회복

* **교착 상태 탐지 알고리즘**
* 기본적으로 자원할당 그래프를 사용하여 알 수 있다.
* **Available :** **자원마다 사용 가능한 자원 수를 표시하는 길이가 m인 벡터이다.**
* **Allocation : 각 프로세스에 현재 할당된 각 형태들의 자원 수를 표시하는 n x m 행렬이다**.
* **Request : 각 프로세스의 현재 요청을 표시하는 n x m 행렬이다. Request[i,j]일 때 프로세스 Pi에 필요한 자원 수가 k개라면, 프로세스 Pi는 자원 Rj의 자원을 k개 더 요청한다.**
* **교착 상태 회복 알고리즘**
* 프로세스를 종료하는 방법
* > **1. 교착 상태의 프로세스를 모두 중지**
* > 2. 교착 상태가 **해결될 때까지 한 프로세스씩 중지**
* **자원을 선점하는 방법**
* > 1. 교착 상태의 프로세스가 점유하고 있는 **자원을 선점하여 다른 프로세스에게 할당하며, 해당 프로세스를 일시 정지시키는 방법**
* > 2. **우선 순위가 낮은 프로세스, 수행된 횟수가 적은 프로세스 등을 위주로 프로세스의 자원을 선점한다.**

**<기아 상태>**

* **프로세스가 필요한 자원을 영원히 할당 받지 못해서 계속 기다리고 있는 상태**
* **교착 상태**가 자원을 자유롭게 할당한 결과에 의한 **자원부족**이라면, **기아 상태는 작업이 결코 사용할 수 없는 자원을 계속 기다리는 결과,** 즉 **교착상태를 예방하려고 자원을 할당할 때 발생하는 기다림의 결과이다.**
* 식사하는 철학자 해결법
* 철학자 4명만 테이블에 동시에 앉도록 한다.
* **철학자가 양쪽 포크를 모두 사용할 수 있을 때 포크를 집을 수 있도록 허용한다. (임계 영역안에서 해야함)**
* 비대칭 해결법을 사용한다. 즉, 홀수번째 철학자는 **왼쪽 포크를 집은 후 오른쪽 포크를 집게하고, 짝수 번째 철학자는 오른쪽 포크를 집은 후 왼쪽 포크를 집게 한다.**
* 기아상태의 해결은 **먼저 기다리는 작업을 발견하고 각 작업이 기다린 시간을 조사, 추적해야 한다.** 또 시스템은 **기아 상태를 발견하면 즉시 새로운 작업의 시작을 대기하도록 조치해야 한다.**
* 단, 빈번한 시스템 대기로 처리량이 감소할 수 잇으므로 매우 신중한 접근이 필요하다.

<스케줄링의 이해>

1. 스케줄링의 개념

* 여러 프로세스가 번갈아 사용하는 자원을 어떤 시점에 어떤 프로세스에 할당할지 결정하는 것.

1. 스케줄링의 목적

* 자원 할당의 **공정성**
* 단위시간당 **처리량 최대화**
* 적절한 반환시간 보장
* 예측 가능성 보장
* **오버헤드 최소화**
* 자원 사용의 **균형 유지**
* 반환시간과 자원의 활용 간에 균형 유지
* **실행 대기 방지 : 에이징 방법으로 해결**
* **우선순위 : 우선순위가 높은 프로세스를 먼저 실행**
* 서비스 사용 기회 확대
* 서비스 수 감소 방지

1. 스케줄링의 기준 요소

* 프로세스 버스트 : 프로세스를 프로세서에서 실행할 때
* 입출력 버스트 : 프로세스가 추가로 실행하려고 입출력을 기다리고 있을 때

1. 스케줄링의 단계

* **작업 스케줄링 : 작업 선택, 장기 스케줄링**
* 시스템 **자원을 사용할 작업을 결정**하는 작업 스케줄링.
* 작업 승인과 프로세서 결정 스케줄링 : **사용 권한 부여 , 중기 스케줄링**
* 프로세서를 **사용할 권한을 부여할 프로세스를 결정하는 작업승인과 프로세서 할당 스케줄링**
* **프로세서 할당 스케줄링 : 준비 상태의 프로세스에 프로세서 할당(디스패칭), 단기 스케줄링**
* 디스패처가 준비상태에 있는 **프로세스 중에서 프로세서를 할당할 프로세스를 결정하는 프로세스 할당 스케줄링이다.**

1. 스케줄링 큐

* 준비 큐에서 프로세스를 하나 선택하여 프로세서를 할당한다.

**<가상 메모리>**

- 가상 메모리를 사용하는 이유는 사용 중이 프로세스 전체가 메모리에 올라와 있을 필요가 없기 때문이다. 즉 **프로세스의 필요 부분만 메모리에 올리고**, 나머지 부분은 다른 프로세스의 필요부분을 올림으로써, 여러 프로세스에 대한 데이터를 메모리에 올려 사용하는 방법이다.

- 장점

> 1. 프로그램 실행시 **물리메모리의 여유공간을 고려하지 않아도 된다**.

> 2. **더 많은 프로그램을 동시에** 메모리에 올릴 수 있다.

> 3. 한번에 올리는 **데이터의 양이 적기 때문에 HDD와 메모리의 I/O 작업속도가 빨라진다.**

- 가상메모리 구현 방법으로는 2가지 방법이있다.

1**. Demand paging(요구 페이징)**

> 가상메모리의 구현 철학과 일치하는 방법으로, **프로그램의 필요부분만 메모리에 올리는 방법이다.**  이렇게 **메모리에 올라온 데이터를 페이지라고 한다.**

> CPU는 논리주소를 이용하여 메모리에 접근하고, 중간에 **MMU(Memory Management Unit)이 물리주소로 변환해준다.** 하드웨어를 이용한 변환이 더욱 빠르기 때문이다. 이 MMU를 Page Table이라고 한다.

> MMU에서는 valid bit을 사용한다.

>> case **“Valid”** : Page Table의 해당 인덱스에 접근이 가능하고, 그 인덱스가 **가리키는 메모리의 물리적 주소에 데이터가 올라와 있다는 것이다**

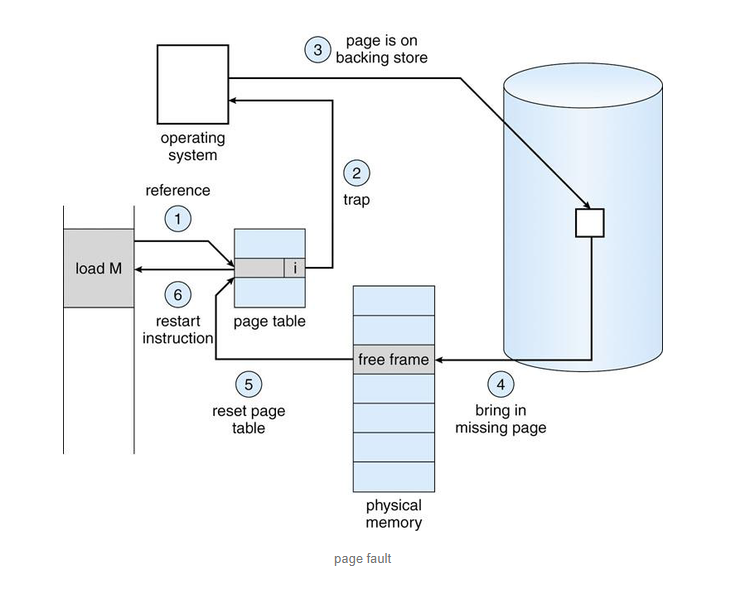
>> case “Invalid” : Page Table의 해당 인덱스에 접근이 불가능 하거나, 해당 메모리 주소에 데이터가 올라와 있지 않다는 것이다. 즉 필요 데이터가 HDD에 존재한다는 것이다.

**- Page Fault :** 지금 실행시켜야할 페이지가 메모리에 올라와 있지 않은 상태를 말한다.

> 페이지 폴트가 발생하면 CPU는 OS에게 알리고 OS는 해당 작업을 잠시 중지시킨다.

> HDD에서 해당 페이지를 메모리에 올린 후 Page Table의 해당 인덱스의 Valid bit을 True로 바꾸어준다.

> 다시 실행한다.



**- Page Replacement :** 페이지 폴트가 발생했는데, **메모리에 공간이 없어서 페이지를 가져올 수 없을 경우**에 사용하는 방법으로, 메모리에 올라와있는 **특정 프레임(page를 가지고 있는 공간)를 정해 메모리에서 삭제 시킨 후 공간을 만들어서 HDD에서 필요 페이지를 가져오는 방법이다.**

> 페이지 교체를 위해서는 **Victim Frame** 즉, 삭제될 Frame을 결정해야 한다. 여기에는 여러가지 방법이 존재한다. 이때 사용하는 방법은 **Modify Bit**을 이용하는 것이다.

>> Modify bit : 메모리에 올라와 있는 page 중에 내부 데이터가 바뀌었는지 알려주는 Bit이다. 내부 **데이터가 바뀌었다면 삭제시 HDD와의 동기화를 위해 Swap-out** 즉, 변한 데이터가 HDD에 업데이트 되는 과정이 필요하다. 그러나 **swap-in/out은 많은 비용을 발생시키므로, Modify Bit ==0 인 페이지를 Victim Frame으로 정한다.** (내부 데이터가 수정되어있지 않으면 HDD에 업데이트 할 필요가 없기 때문이다.)

> 과정을 설명하면 다음과 같다.

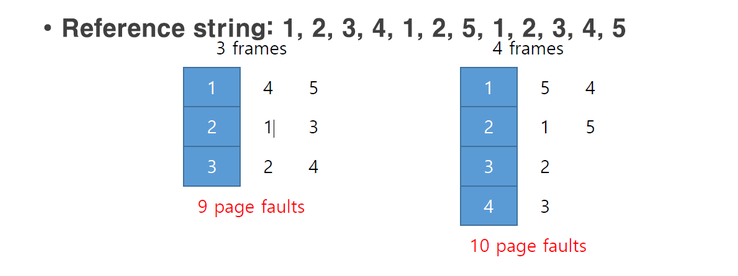
>> 1. CPU에서 필요한 페이지를 메모리에서 찾는다. 만약 해당 페이지가 없고, 메모리에 모든 Frame이 가득 찼다면 Modify Bit을 확인한다.

>> 2. 확인 후 Victim Frame을 정하고, 해당 Frame과 필요한 페이지를 교환한다.

>> 3. Page Table의 인덱스를 갱신한 후 명령어를 실행한다.

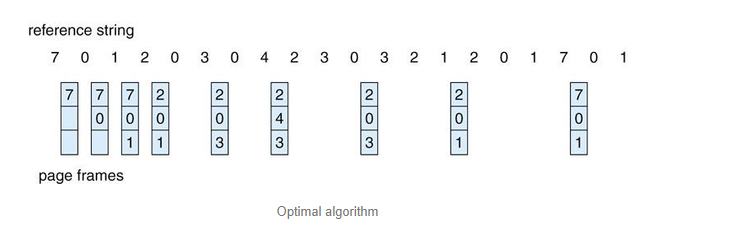
> 페이지 교체에 사용되는 알고리즘은 다음과 같은 것들이 있다.

>> 1. FIFO Algorithm : 말 그대로 메모리에 올라와 Frame을 차지한지 **가장 오래된 Frame을 Victim으로 설정하는 것이다.** 하지만 만약 지워진 페이지가 자주 사용하는 페이지라면 Frame의 전체 크기와 관계 없이 Page Fault가 자주 발생하게 된다.



>> 2. Optimal Algorithm : 가장 **오랫동안 사용되지 않을 Frame을 Victim Frame으로 결정하**는 것이다. 하지만 이 방법은 말그대로 예측이기 때문에, 이론적으로만 존재한다.

밑에 그림을 참고해보면 Page Fault가 발생했을 때 7번 프레임이 삭제된다. 그 이유는 이 후에 사용할 페이지를 탐색했을 때**, 7번이 다른 것보다 가장 늦게 사용되기 때문이다.**



>> 3. LRU (Least Recently Used) Algorithm : 가장 **오랫동안 사용되지 않은 프레임을 Victim Frame으로 결정하는 방법이다.** 이 알고리즘의 근거는 최근에 사용되면 될수록 다시 참조할 가능성이 크다는 것에 전재한다.

>> 구현에는 2가지 방법이 있는데, 첫번째로 해당 page가 실행될 때, 실행 된 시간을 page table의 해당 부분에 저장하는 방법이고, 두번째는 실행순서를 Stack으로 관리하는 방법이다. 하지만 이 방법들은 HW지원이 필요하고, 딱 봐도 효율이 안 좋다.

>> HW 가 필요없는 방법으로 LRU-Approximation Algorithm이 있다. 이 방법은 page table내에 각 페이지마다 8bit을 할당하고**, Access 될때마다 차상위 비트에 1을 첨가한다. 그리고 일정시간마다 오른쪽으로 시프트연산을 한다.** Fault 발생 시 가장 큰 값을 가진 페이지가 가장 최근에 사용된 page가 된다. 그러므로 가장 작은 값을 가진 page가 victim이 된다.

>> Second-Change Algorithm : 이 방법은 page table에 reference bit라는 1bit을 추가하는 것이다. FIFO를 기반으로 하지만, **Access가 발생하면 해당 page table의 page의 bit을 1로 바꾸어 준다. Fault 발생시 FIFO 기반으로 가장 처음에 들어온 page가 victim이 되지만, reference bit == 1 이라면 0으로 초기화 한 후 다른 victim을 찾음으로써 second-chance를 주는 것이다.**

>> LFU – MFU (Least,Most Frequently Used) Algorithm : page table에 각 페이지에 Access 횟수를 저장함으로써, **가장 적게 사용된, 혹은 가장 많이 사용된 page를 victim으로 하는 방법이다.** 각각의 철학은 가정 적게 사용되니까 다음에도 사용 안될것이다! 와 가장 많이 사용됐으니까 이제 사용 안되겠지! 이다.

**> Frame Allocation :** 프로세스가 사용할 page만 메모리에 올려야 하는데(Demand Paging), 이**때 Frame을 할당해주는 방법이 있다.**

>> **1. Fixed Allocation :**

>>> Equal Allocation - 모든 프로세스에게 고정된 양의 Frame을 할당하는 것이다.

>>> Proportional Allocation - 프로세스의 사이즈에 비례하여 Frame을 할당해 주는 것이다.

>> 2. Priority Allocation : 우선순위가 높은 프로세스에 더 많은 Frame을 할당하는 것이다. **프레임을 많이 할당하면 그만큼 page fault가 줄어듬으로써 작업속도가 향상된다.** 즉, 우선순위가 높은 **프로세스를 빠르게 처리하고 Frame을 반환하여 Frame 순환율을 높이는 것이**다.

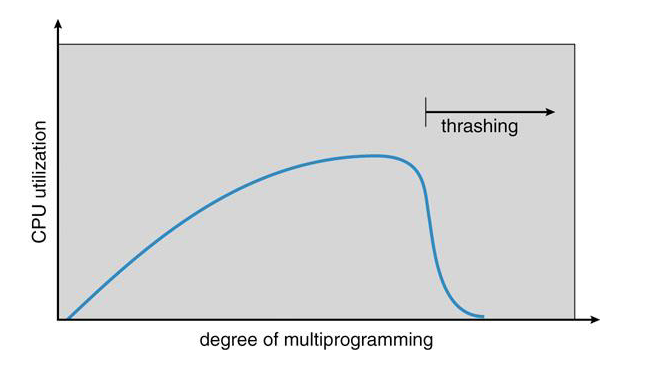
>> 3. Global Replacement : Victim Page를 찾을 때 모든 프로세스의 모든 페이지에 대해서 찾는 것을 말한다.

>> 4. Local Replacement : Victime Page를 찾을 때 해당 프로세스의 페이지에서만 찾는 것을 말한다.

**> Trashing :** Multi Programming에서 CPU의 효율성을 높이기 위해 메모리에 여러 프로세스를 올려놓고 각 프로세스가 CPU를 돌아가면서 점유한다. 그러나 프로세스가 많아질수록 **하나의 프로세스가 가지는 메모리 Frame의 양은 적어지고,** 그럼으로써 Page Fault가 많이 발생하게 된다.

> 이를 처리하기 위해 Page Replacement를 할텐데, 이것이 많아지면 실제 CPU처리량 보다 메모리와 HDD간의 I/O 시간이 더 많아지게 된다. **그럼으로써 CPU는 효율성이 떨어지게 된다.**

**> 더 큰 문제점은 이를 해결하기 위해서 CPU는 더 많은 프로세스를 메모리에 올리게 된다. 결국 악순환의 반복이 되며, 프로그램은 비정상적인 종료를 한다. 이것이 Trashing이다.**



>> Trashing 해결방법 : 해결하려면 각 프로세스 실행때마다 필요로하는 Frame의 양을 할당해 주어야 하는데, 매 실행마다 필요 Frame을 아는 것은 불가능하다. 그러므로 **예측을 해야한다.**

>>> Locality 로 예측가능하다. 지역성을 기반으로 가장 많이 사용하는 page를 메모리에 고정시켜 놓는 것이다. Victim Frame이 되지 않도록.

2. Demand Segmentation

**To Do: 에이징, 세그멘테이션, 메모리메니지먼트**