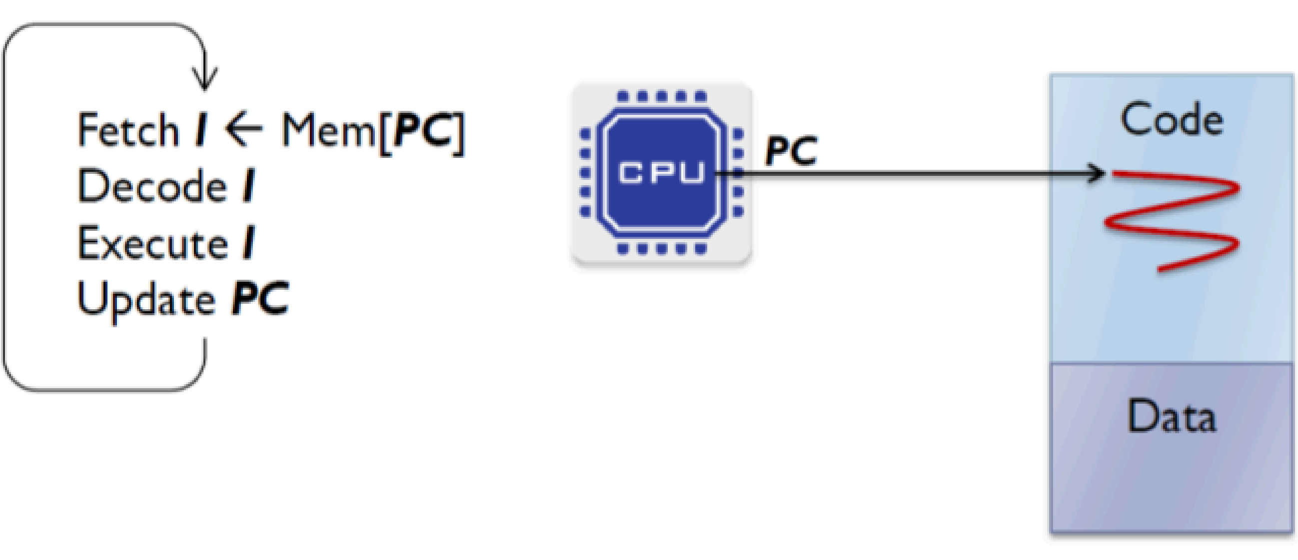
Process 2

CPU가 프로세스를 관리하는 방법 4가지

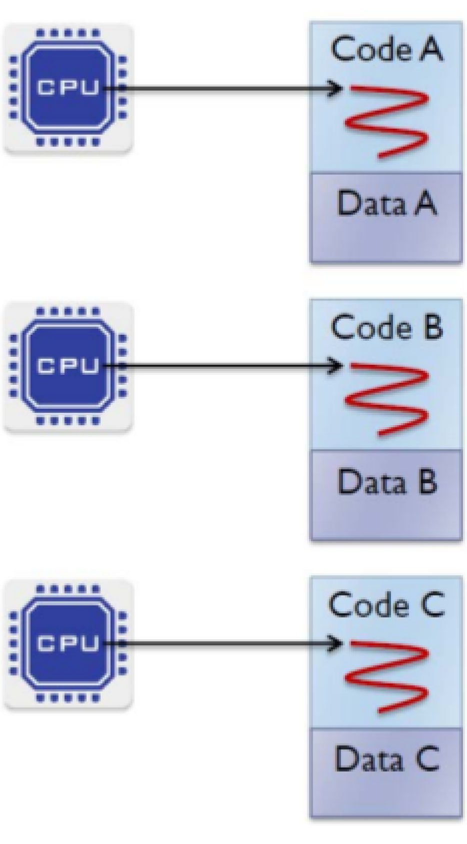
1. Running a Process

- PC는 메모리에 있는 다음에 실행될 명령어를 가르킴
- CPU는 PC가 가르키는 명령어를 fetch, decode, execute, update



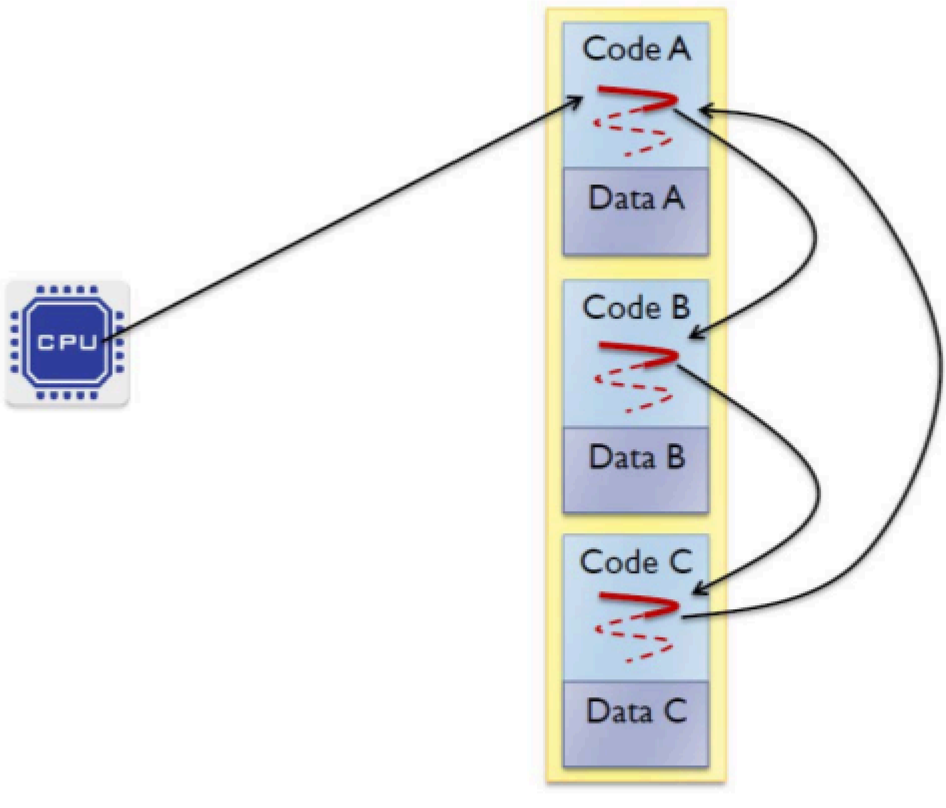
2. Running Multiple Processes

- 위 작업을 여러 CPU가 수행



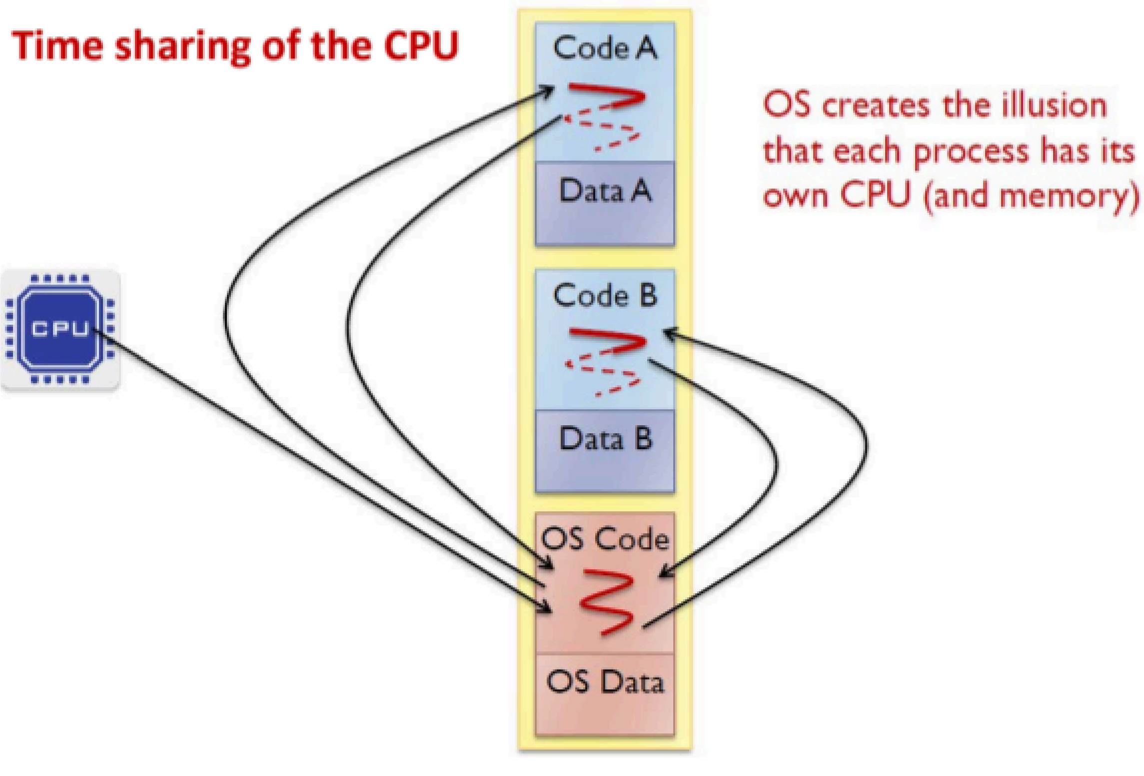
3. Interleaving Multiple Processes

- 하나의 CPU는 하나의 프로세스만 담당하는 것이 아님
- 여기저기 번갈아가며 프로세스의 명령어들을 fetch, decode, execute, update



4. virtualizing the CPU

- CPU time sharing
  - OS 또는 하이퍼바이저가 각 프로세스에 일정한 시간 부여
  - 시간이 다 되면 context switch
- 각 프로세스가 마치 각자의 CPU와 메모리를 가진 듯한 illusion
- CPU를 여러 개로 나누어 각 프로세스에 가상 CPU를 제공
- os code가 code A, code B에 관여하는 모습



CPU를 효과적으로 가상화해서 제어하는 방법?

- OS는 물리적인 CPU를 time sharing으로 share 해야 함.
- issue
  - 성능 : 시스템에 과도한 오버헤드를 주지 않고 가상화를 구현할 수 있을까?
  - 제어 : CPU 제어권을 유지하면서 효과적으로 프로세스를 실행할 수 있을까?
    - CPU가 한 프로세스에 독점되지 않도록, 무한루프에 빠지지 않도록

Direct Execution

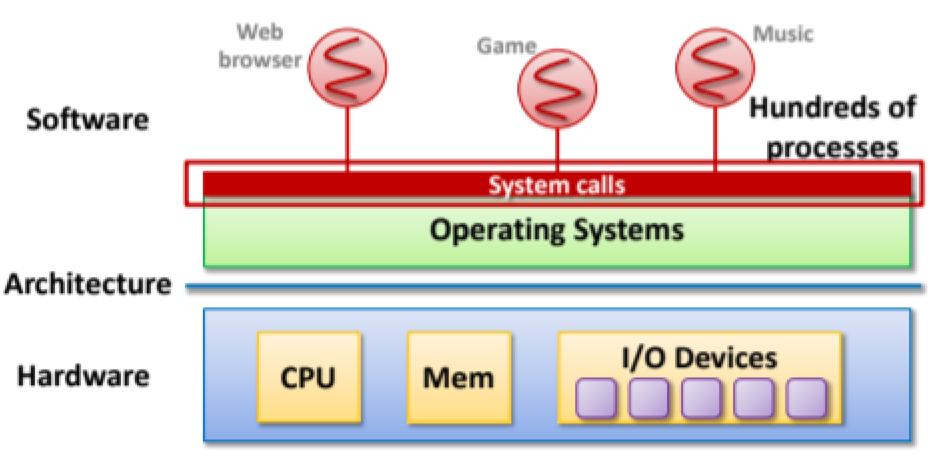
- 제약조건이 없는 경우 (no restriction)
- CPU에서 직접 프로그램 실행해버리기
- OS
  1. process list로 들어가는 entry 생성
  2. 프로그램에 메모리 할당
  3. 프로그램을 메모리에 로딩
  4. argc/argv로 스택 만들기
  5. 레지스터 비우기
  6. 메인함수 execute
     - Program
       7. 메인함수 run
       8. 메인함수 return
  9. 프로세스 메모리 비우기
  10. process list에서 삭제
- 프로그램을 실행하는 데 있어서 제한이 없다면 OS는 어떤 것도 제어할 수 없으며 단지 라이브러리로서 존재하게 됨

Problem 1 : Restricted Operation

- 만약 프로세스가 restricted operation을 수행하려 한다면?
  - 디스크에 IO 요청
  - CPU나 메모리 같은 시스템 자원에 접근 요구
- 방법 : protected control transfer 사용
  - user mode : app들은 하드웨어 자원에 full access를 갖지 못함
  - kernel mode : OS는 머신의 모든 자원에 대한 권한을 가짐

System call

- kernel이 user program에 다음과 같은 특정 기능들을 조심히 제공
  - file system에 accessing
  - 프로세스 creating, destroying
  - 다른 프로세스들과 communicating
  - 더 많은 메모리 할당
  - etc



trap instruction

- user mode에서 kernel mode로 전환하는 명령어
- user program 실행을 중단, OS의 kernel code로 jump
- CPU의 privilege level을 상승시켜 kernel mode로 진입

return from trap instruction

- user program의 calling으로 return
- user mode로 돌아가기 위해 privilege level을 감소시킴

trap은 OS에서 어떤 코드가 실행되는지 어떻게 알 수 있을까

- trap handler
  - trap이 발생했을 때 실행되는 코드
  - trap은 system call일 수도, page fault같은 것일 수 도 있음
  - 보통 시스템 콜 처리, 예외 처리, 인터럽트 처리 등을 담당
- trap table
  - trap 번호와 이를 처리할 trap handler의 주소가 매핑된 테이블
  - trap handler가 trap table을 참조
  - 인터럽트도 이와 비슷하게 (사실 얘가 원조긴 함) interrupt handler가 interrupt vector table을 참조

System call number

- 각 시스템 콜 마다 할당된 번호
- user code는 레지스터에 원하는 system call number를 넘겨줘야 함

Limited Direction Execution Protocol

- OS 부팅 시 trap table 초기화
- CPU는 시스템콜 핸들러의 주소를 기억(?)
- OS가 실행 중
  - process list에 해당 process 추가
  - 프로그램 메모리 할당, 적재
  - argv로 user stack 셋업, kernel stack에 reg/PC 저장
  - trap에서 return
- CPU 차례
  - kernel stack에서 레지스터 restore
  - 유저모드로 전환 후 메인으로 점프
- Program
  - 메인함수 실행
  - 시스템 콜 호출
  - OS로 trap into
- CPU
  - kernel stack에 레지스터 저장, 커널 모드로 진입
  - trap handler로 jump
- OS
  - trap handler가 trap 처리
  - trap은 사실 시스템 콜, 현재 시스템 콜 작업 처리
  - trap에서 return
- CPU
  - kernel stack에서 레지스터 복구
  - 유저모드로 진입 후 다음 PC로 jump
- 프로그램
  - 계속 실행하다가 메인에서 return
  - exit()로 다시 트랩 발생
- OS
  - 프로세스의 메모리 해제
  - process list에서 삭제

Problem 2 : Switching Between Processes

- OS는 프로세스를 스위칭 할 수 있도록 어떻게 CPU 제어권을 되찾을까?
  - **cooperative approach** : 시스템 콜 대기
    - 프로세스들이 주기적으로 CPU 제어권을 잃음
    - 시스템 콜 yield로 프로세스가 CPU 제어권 양보
    - divide by zero, 접근하면 안되는 메모리를 접근하는 경우처럼 app이 스스로 OS에 제어권을 넘겨주기도 함
    - 근데 이 경우 프로세스가 무한루프에 걸리면 리붓해야함
    - 현대 OS에선 사용 x
  - **non - cooperative approach** : 그냥 OS가 제어권을 가져감
    - timer interrupt
    - 부팅할 때 OS가 timer를 작동시킴
    - timer가 milliseconds 단위로 interrupt를 raise
    - 인터럽트 발생 시 :
      - 현재 실행 중인 프로세스를 halt
      - program의 state를 save

    - 미리 설정해둔 인터럽트 핸들러가 실행됨

Process 2

  - timer interrupt 덕에 OS는 CPU의 제어권을 다시 가질 수 있음

Context

Saving and Restoring Context

- scheduler는 다음과 같은 행동을 결정
  - 현재 프로세스를 계속 실행시킬지, 다른 프로세스로 전환할지
  - 만약 전환하기로 결정했다면 OS는 context swtich 수행

Context Switch

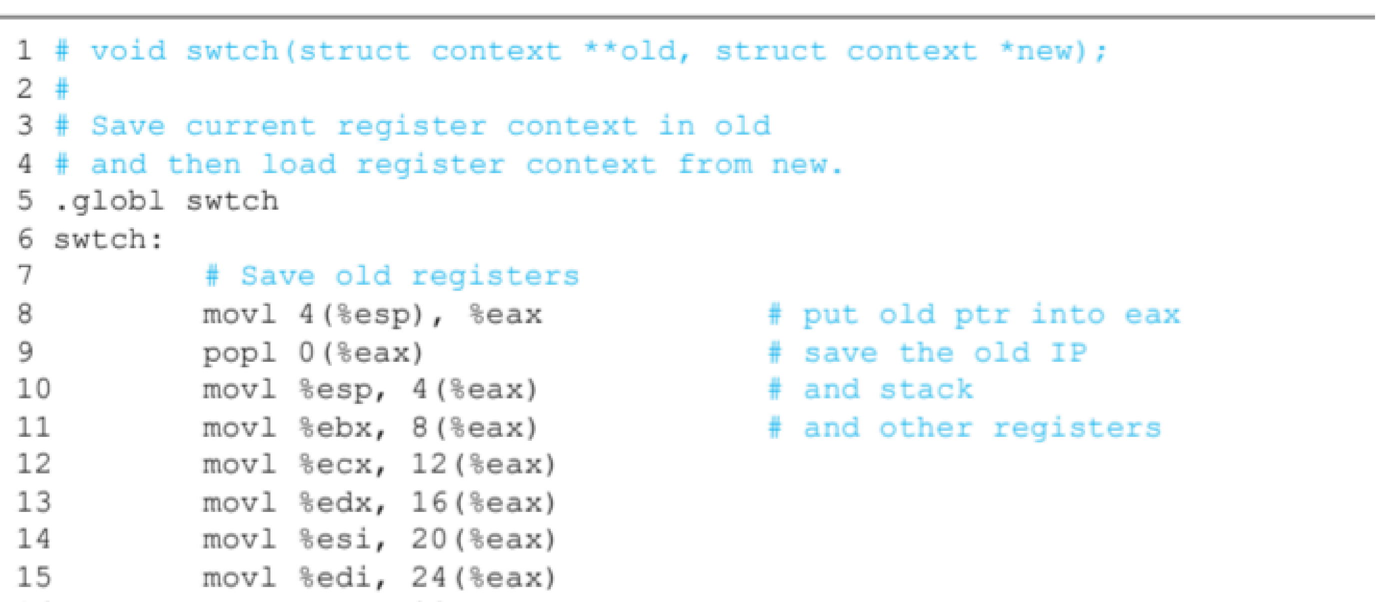
- 어셈블리 수준에서 매우 빠르게 처리
- kernel stack에 현재 프로세스와 관련된 몇몇 register value들을 저장
  - general purpose registers : 프로세스가 계산에 사용한 값들 저장
  - PC
  - kernel stack pointer : 현재 프로세스의 커널 스택 위치 저장
- kernel stack으로부터 곧 실행될 프로세스의 regsiter value들을 restore
- 곧 실행될 프로세스의 kernel stack으로 스위칭

Limited Direction Execution Protocol (Timer interrupt)

- OS
  - OS를 부팅하며 trap table 초기화 (미리 메모리에 올려놔서 access speed 상승)
- CPU
  - CPU가 시스템 핸들러랑 타이머 핸들러 주소 저장
- OS
  - interrupt timer 시작
- CPU
  - timer 작동
  - 몇 ms 뒤 CPU interrupt
- Program
  - 프로세스 A 실행
- CPU
  - timer interrupt
  - kernel stack에 프로세스 A 상태 저장
  - 커널모드로 변경
  - trap handler(interrupt handler와 혼용되는 듯...)로 jump
- OS
  - trap 처리
  - switch() 루틴 호출
  - 프로세스 A 상태를 PCB (process control block = proc-struct)에 저장
  - proc-struct(B)에서 프로세스 B 상태(레지스터들) restore
  - 프로세스 B의 kernel stack으로 스위칭
  - trap에서 return
- CPU
  - 커널 스택에서 프로세스 B의 레지스터들 restore
  - 유저 모드로 전환
  - B의 PC로 jump
- program
  - 프로세스 B 수행

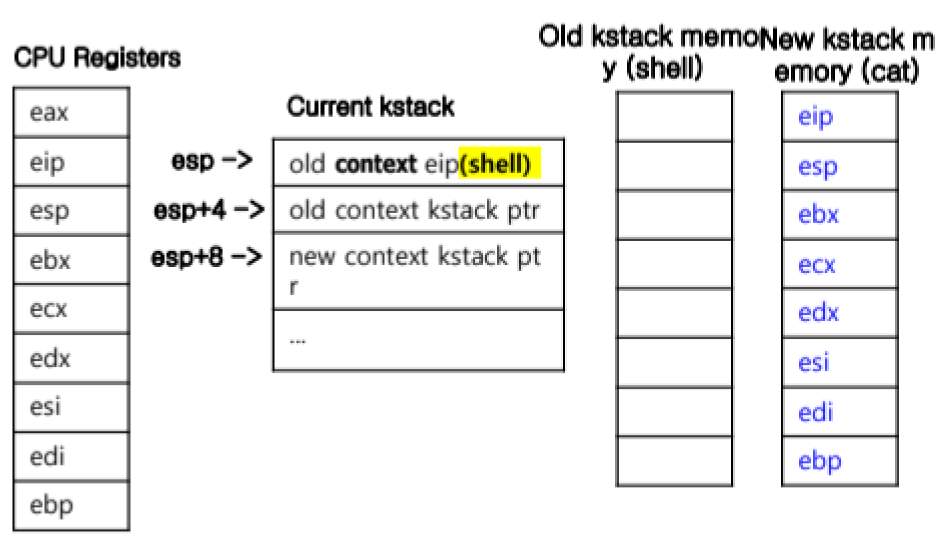
xv6 Context Swtich Code

- 유닉스 기반 경량 운영체제
- kernel stack에 프로세스의 실시간 상황을 저장
- PCB에는 장기적 메타 데이터를 저장. 이때 kstack pointer도 포함
- 각 프로세스 마다 고유의 kstack이 존재
- full code

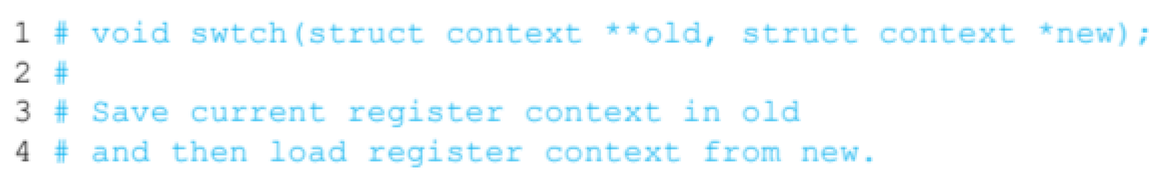


```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
        # Save old registers
        movl 4(%esp), %eax              # put old ptr into eax
        popl 0(%eax)                    # save the old IP
        movl %esp, 4(%eax)              # and stack
        movl %ebx, 8(%eax)              # and other registers
        movl %ecx, 12(%eax)
        movl %edx, 16(%eax)
        movl %esi, 20(%eax)
        movl %edi, 24(%eax)
        movl %ebp, 28(%eax)

        # Load new registers
        movl 4(%esp), %eax              # put new ptr into eax
        movl 28(%eax), %ebp             # restore other registers
        movl 24(%eax), %edi
        movl 20(%eax), %esi
        movl 16(%eax), %edx
        movl 12(%eax), %ecx
        movl 8(%eax), %ebx
        movl 4(%eax), %esp              # stack is switched here
        pushl 0(%eax)                   # return addr put in place
        ret                             # finally return into new ctxt
```



상세 분석



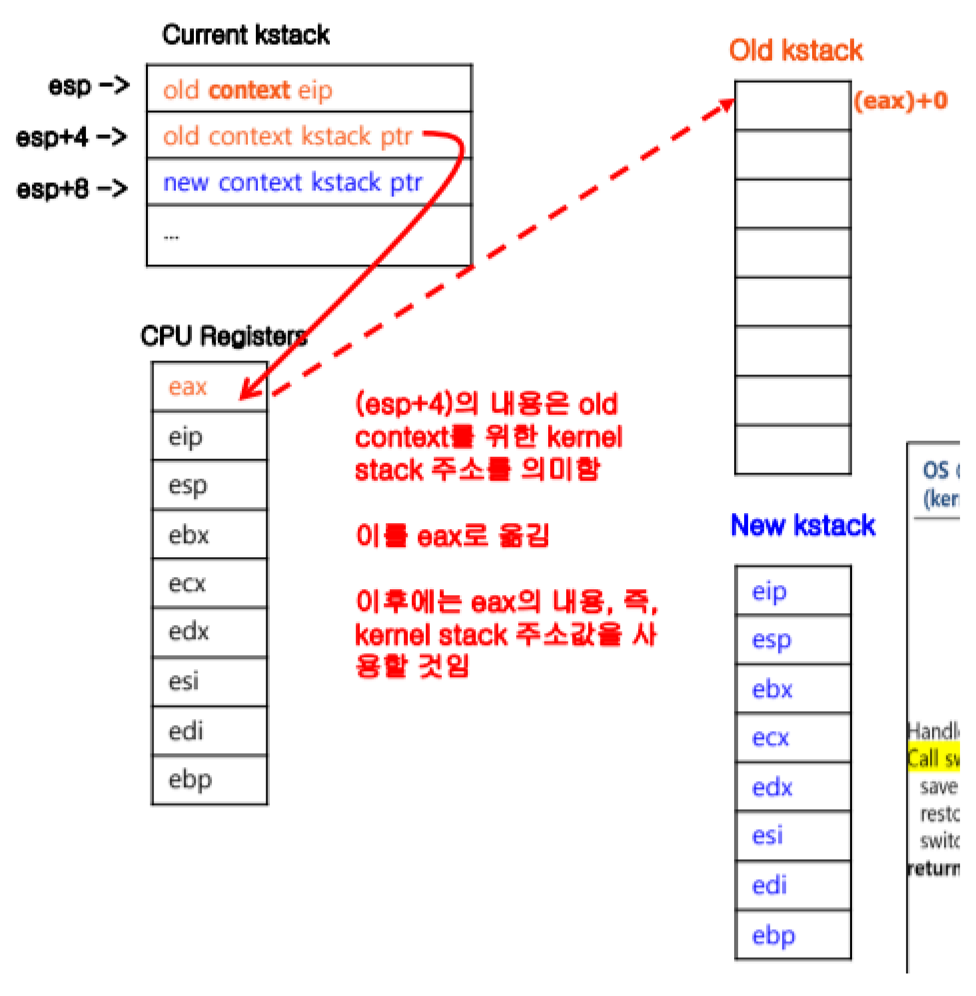
```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
```

- old : 현재 실행 중인 프로세스의 context를 저장할 메모리 주소
- new : 새로운 프로세스의 context가 저장된 주소

1. 현재 context PCB 시작 주소를 eax에 저장

```
swtch:
        # Save old registers
        movl 4(%esp), %eax              # put old ptr into eax
```

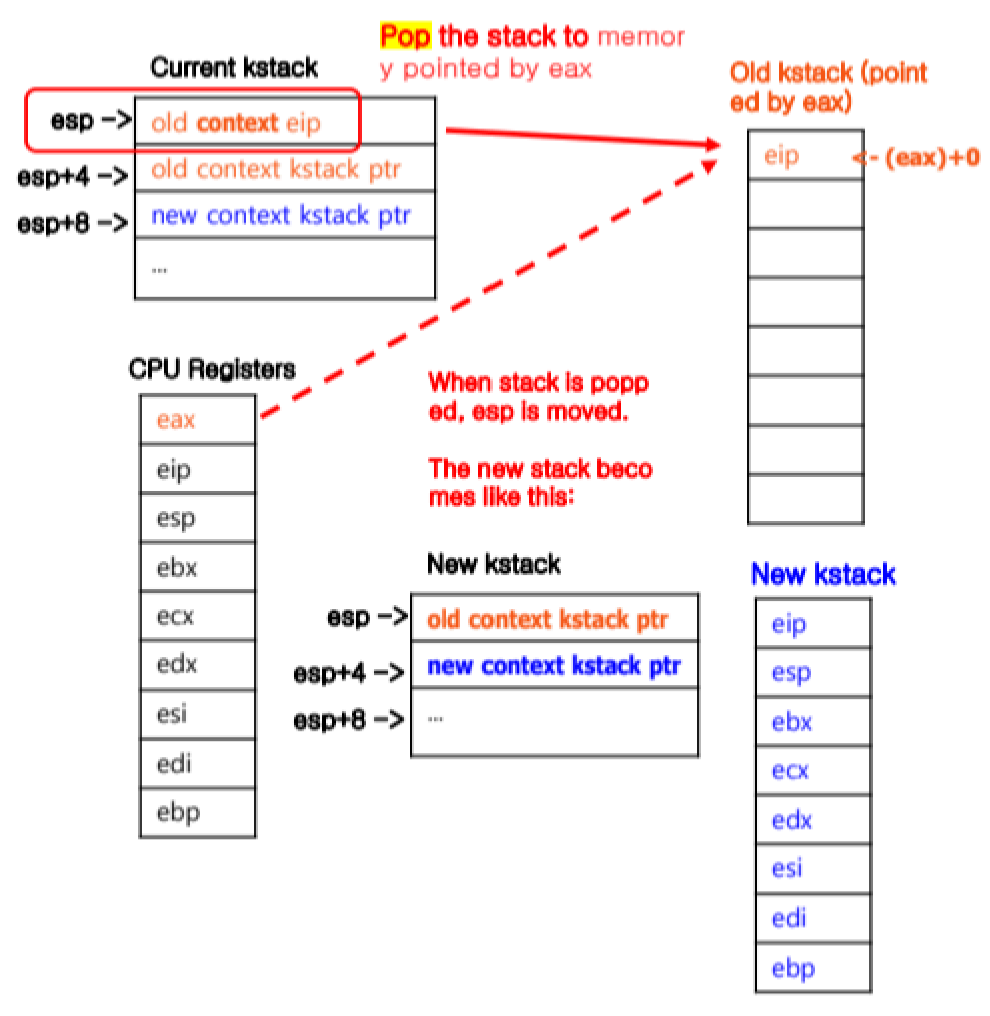
- esp는 스택포인터
- 0(%esp)에는 caller가 복귀할 주소가 담겨 있음
- 4(%esp)에는 현재 context PCB 주소가 담겨 있음
- 8(%esp)에는 새로운 context PCB 주소가 담겨 있음
- => 현재 context가 저장된 PCB의 시작 주소를 eax에 저장



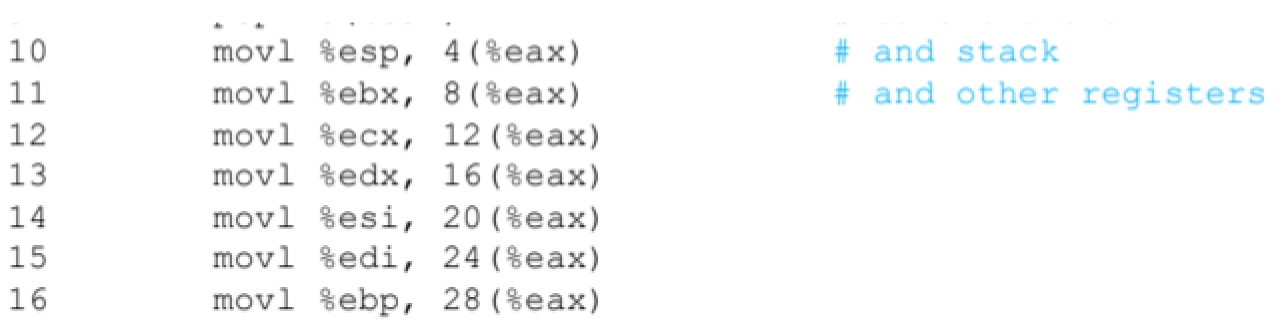
2. 현재 context PCB의 맨 앞에 sp 저장

```
        popl 0(%eax)                    # save the old IP
```

- 스택의 맨 위 값을 꺼내서 0(%eax)에 저장.
  - 이때 스택 맨 위 값은 0(%esp)로 caller가 복귀할 주소가 담겨 있음.
  - caller는 swtch 함수를 콜한 스케쥴러이며 스케쥴러가 복귀할 주소는 프로세스 A가 스위칭 되기 전 마지막 instruction의 다음 instruction 주소임
- 스택 포인터를 4 옮김 (이제 8(%esp)를 가르키게 됨)
  - 만약 주소 A가 eax에 저장되어 있으면 0(%eax)는 A[0]
- => 현재 context PCB의 첫 위치에 다음 instruction 주소 저장하고 스택++



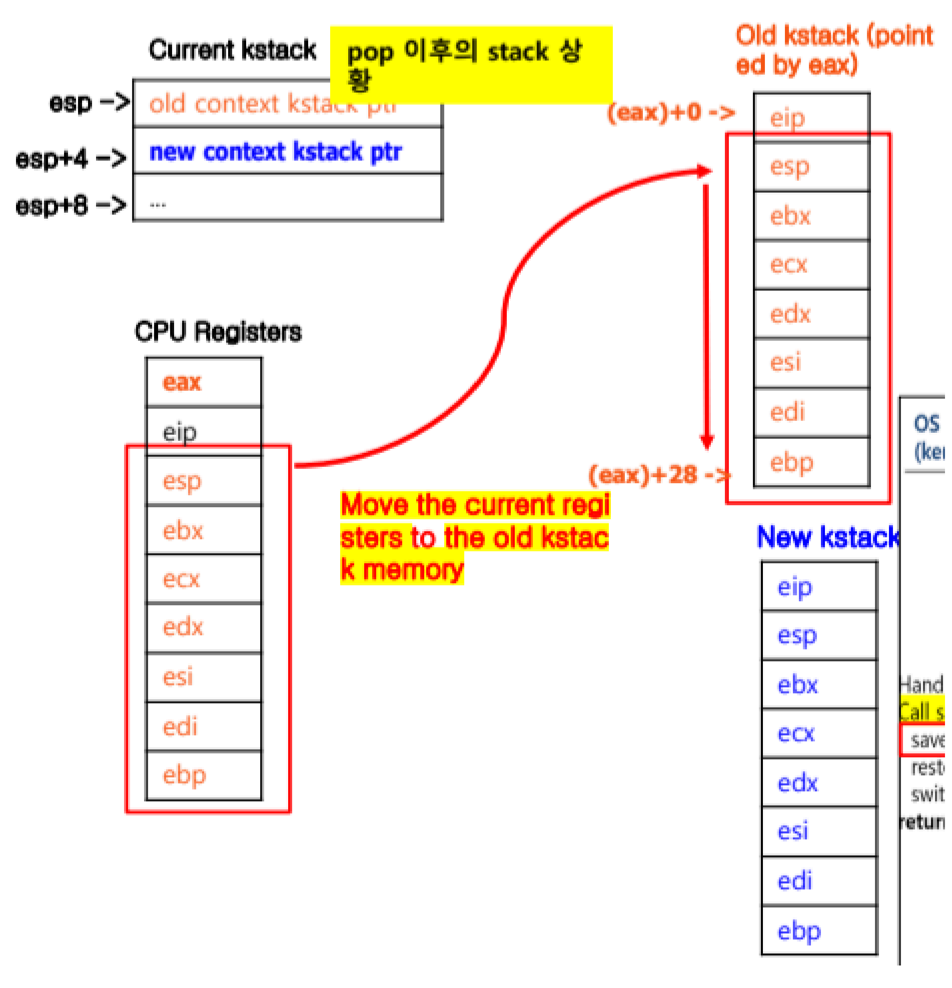
3. 현재 context PCB에 나머지 레지스터 저장



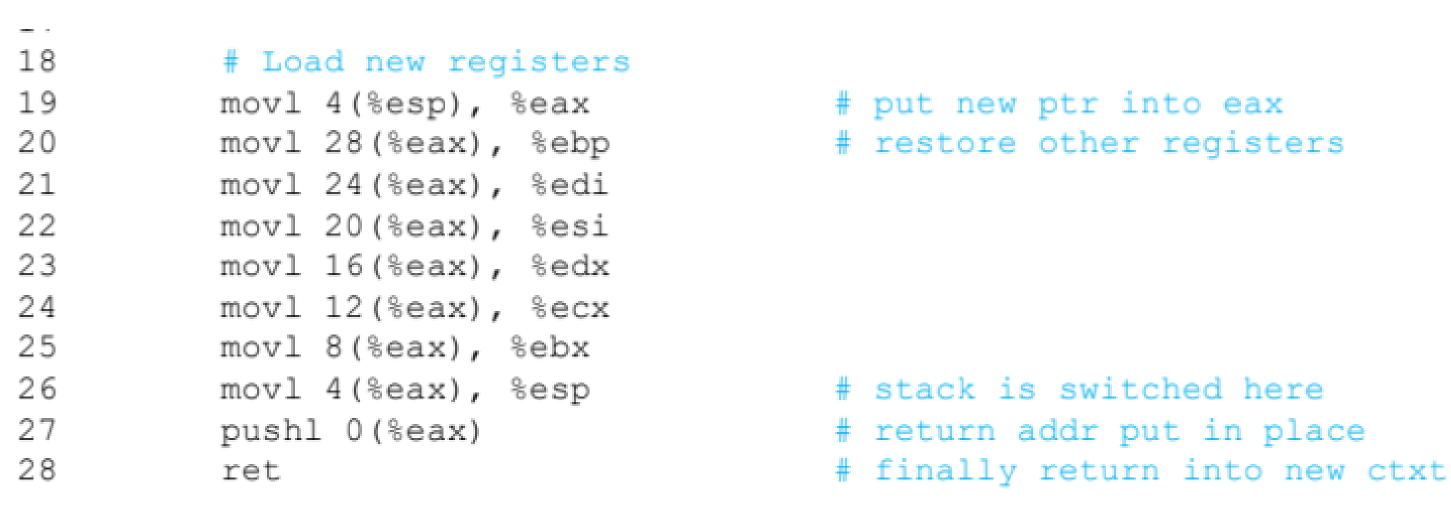
```
        movl %esp, 4(%eax)              # and stack
        movl %ebx, 8(%eax)              # and other registers
        movl %ecx, 12(%eax)
        movl %edx, 16(%eax)
        movl %esi, 20(%eax)
        movl %edi, 24(%eax)
        movl %ebp, 28(%eax)

```

- 각 프로세스는 독립적인 스택을 가짐
- 각자의 스택 위치를 저장해야함
  - 현재 context PCB의 첫 위치엔 ip를 저장하고 그 다음은 sp를 저장
- 그 후 나머지 레지스터들도 현재 context PCB에 줄줄이 저장



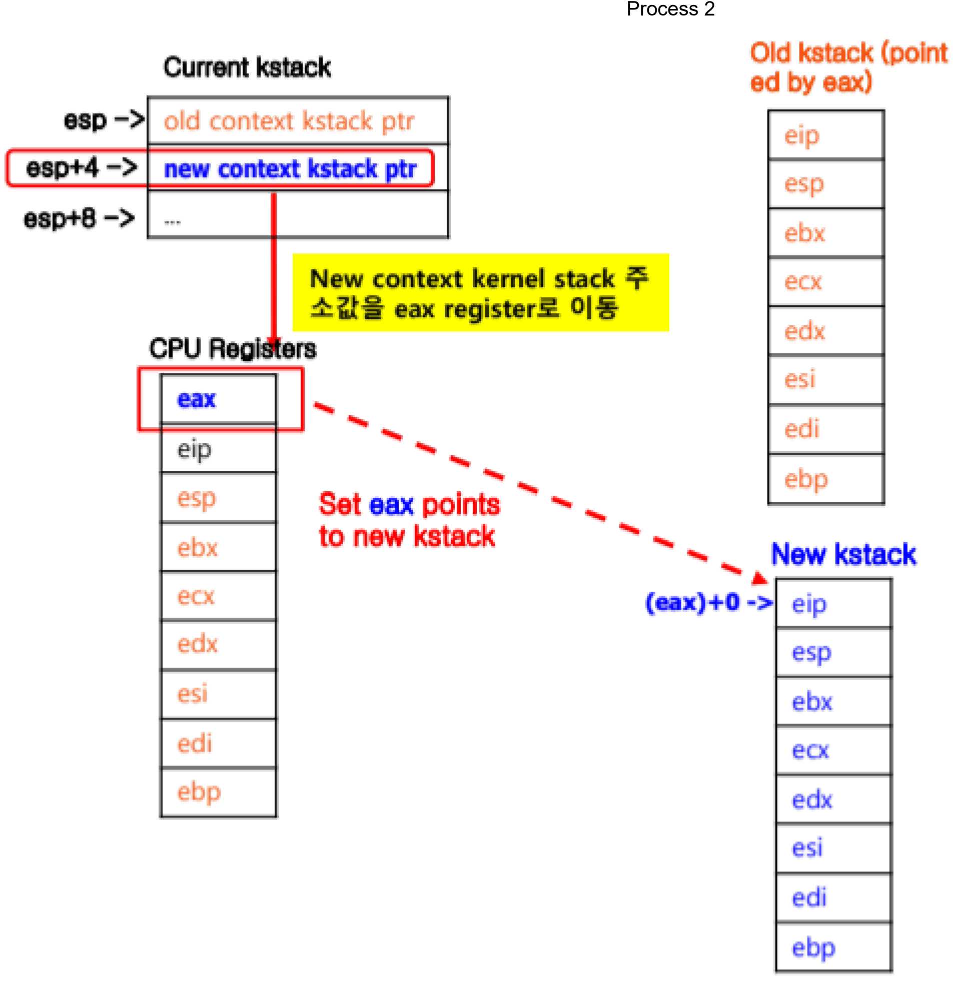
4. 새로 실행할 context PCB (반복)



```
        # Load new registers
        movl 4(%esp), %eax              # put new ptr into eax
        movl 28(%eax), %ebp             # restore other registers
        movl 24(%eax), %edi
        movl 20(%eax), %esi
        movl 16(%eax), %edx
        movl 12(%eax), %ecx
        movl 8(%eax), %ebx
        movl 4(%eax), %esp              # stack is switched here
        pushl 0(%eax)                   # return addr put in place
        ret                             # finally return into new ctxt
```

- 이 다음 4(%esp)는 새로운 프로세스의 context PCB 주소
- 새 context PCB에서 레지스터 값들을 가져와 CPU의 레지스터에 저장
- 스택 포인터도 PCB에 가져와서 esp에 저장

- ip를 스택의 최상단에 올리고 리턴



Worried about concurrency?

- 만약 인터럽트/트랩 핸들링 동안 다른 인터럽트가 발생한다면?
  - 인터럽트 처리 동안 발생한 인터럽트를 disable
  - 내부 데이터 구조를 동시에 접근하는 문제를 막기 위해 수많은 locking schemes를 사용