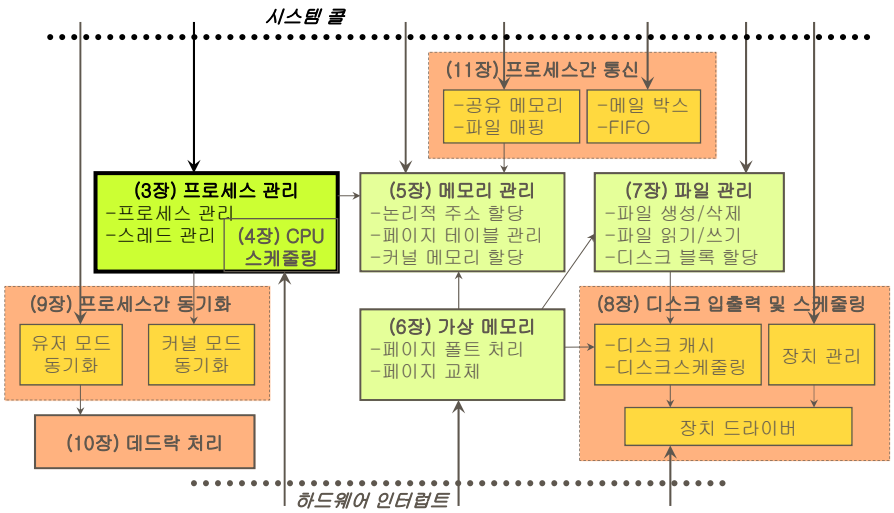
프로세스 관리

- ✓프로세스
- ✓메모리 상의 프로세스 구성
- ✓프로세스의 일생과 상태천이
- ✓프로세스 제어블록
- ✓스케줄링과 문맥교환
- ✓멀티스레드 프로세스
- ✓picoKernel의 문맥교환과 스레드 구현

관련 운영체제 구성 모듈



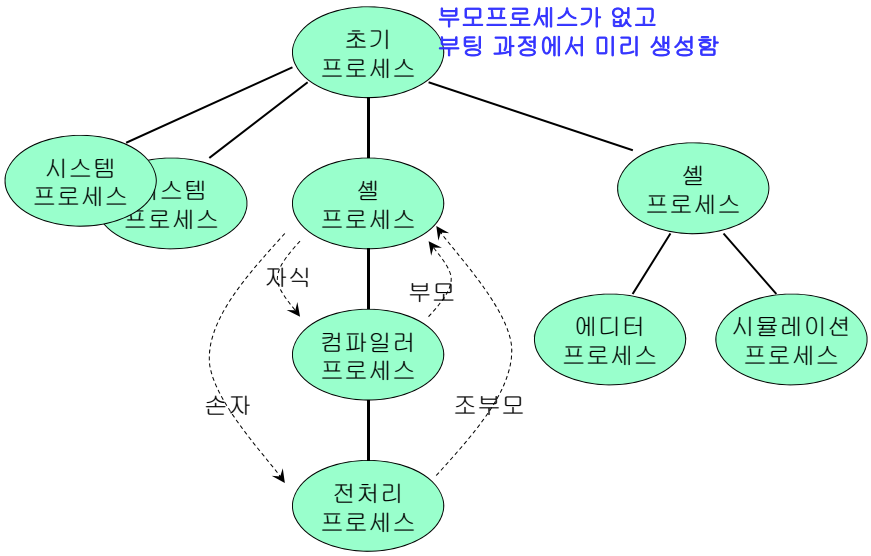
이해하고 넘어가야 할 내용들

- ✓ 문맥교환의 의미 및 이를 위해 처리할 내용
- ✓ CPU가 운영체제 부분과 프로세스들 간에 이동해 가면서 실행하는 과정
- ✓ 프로세스/스레드의 의미 및 이들의 연관성
- ✓ 프로세스/스레드 제어블록이 필요한 이유 및 여기에 기록해야 할 정보들
- ✓ 스케줄링과 문맥교환의 실제 구현 방법
- ✓ 프로세스/스레드의 생성 및 삭제 과정에서 해야할 일들과 이를 구현하는 방법

프로세스

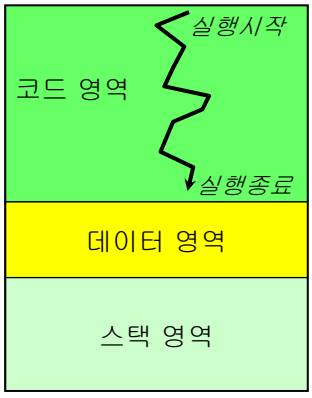
- ✓ 프로세스 (process)
  - ✓ 프로그램에 표현된 내용을 실행하고 있는 능동적 개체
  - ✓ 스케줄링 및 문맥교환의 대상
  - ✓ 시스템 콜 함수를 통해 생성되고 소멸됨
- ✓ 프로세스의 생성
  - ✓ 실행중인 프로세스가 운영체제에 시스템 콜을 통해 새로 생성
  - ✓ 사용자의 명령 해석기 (유닉스의 셸 등) 프로세스는 사용자로부터의 입력을 받아서, 그 내용에 따라 새로 프로세스를 생성하도록 시스템 콜 함수로 운영체제 기능 호출
  - ✓ 모든 프로세스들은 부모-자식의 관계로 하나의 계보 형성
- ✓ 프로세스의 종료
  - ✓ 모든 프로그램은 실행을 완료한 시점에 종료를 위한 시스템 콜 함수를 호출하도록 구성되어 있음
  - ✓ 프로그램의 버그에 의하거나, 다른 프로세스로부터의 강제종료 시스템 콜 함수를 통하여 강제로 종료될 수도 있음

프로세스들 간의 계보

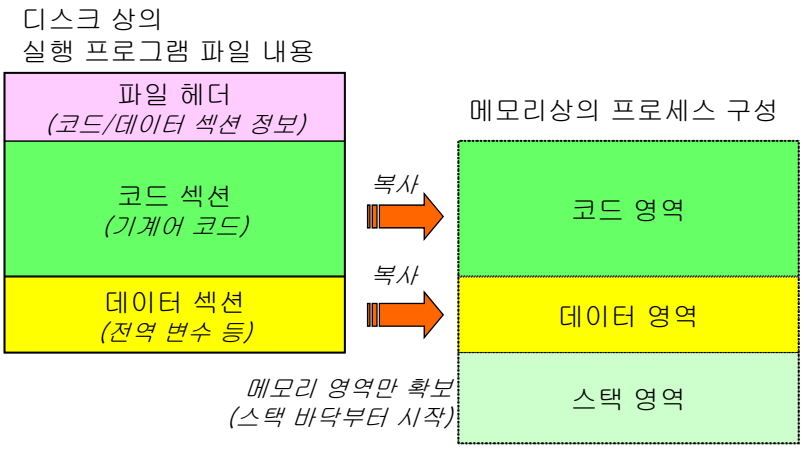


메모리 상의 프로세스 구성 및 실행

- ✓코드 영역:
  - ✓기계어 코드로 이루어진 실행 내용
  - ✓시작위치로부터 기계어 코드에 따라 순차적으로 실행됨
- ✓데이터 영역:
  - ✓전역변수
  - ✓실행 중 할당 받은 메모리 영역 등
- ✓스택 영역:
  - ✓함수 호출과정에서 파라미터 전달이나 반환 주소 기록
  - ✓함수내의 지역변수를 위한 영역 등



실행파일의 적재



유닉스의 프로세스 생성

- ✓ Fork 와 Exec의 2단계로 구성
- ✓ Fork() 시스템 콜
  - ✓ 호출한 프로세스(부모)와 동일한 복제 프로세스(자식)를 생성
  - ✓ 현재까지의 실행 상태에서 2개의 프로세스는 이후에도 실행 내용이 동일함
  - ✓ 일반적으로 자식 프로세스는 다른 프로그램을 실행하도록 변신함
- ✓ Exec() 시스템 콜
  - ✓ 현재 실행 중인 내용을 완전히 버리고
  - ✓ 인수로 지정된 실행파일을 처음부터 실행하는 것으로 변신함
  - ✓ 프로세스의 번호나 부모-자식 관계 등은 그대로 유지됨

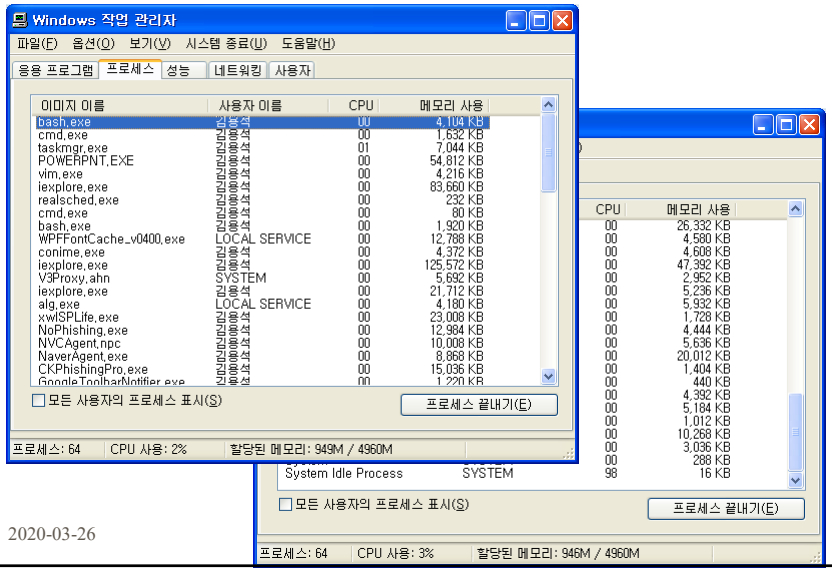
유닉스의 Exec 시스템 콜 함수의 처리

0. 현재 프로세스가 사용하던 메모리는 전부 해제
1. 지정된 실행파일을 열고 헤더 정보를 읽기
   Code 섹션 크기, data 섹션 크기, 시작 주소 등
2. code 섹션 크기만큼의 메모리를 할당하고 여기에 실행파일의 내용을 읽어서 메모리에 적재
3. data 섹션 크기만큼의 메모리를 할당하고 여기에 실행파일의 내용을 읽어서 메모리에 적재
4. 스택을 위한 표준 크기의 메모리를 할당
5. SP 레지스터는 스택 영역 메모리의 바닥 주소로 설정
6. IP 레지스터를 코드영역 메모리의 시작 주소로 설정

(실행하게되면 설정된 IP 주소(시작주소)의 명령어 부터 실행)

Windows 프로세스 목록

실행 창에서 taskmgr 실행



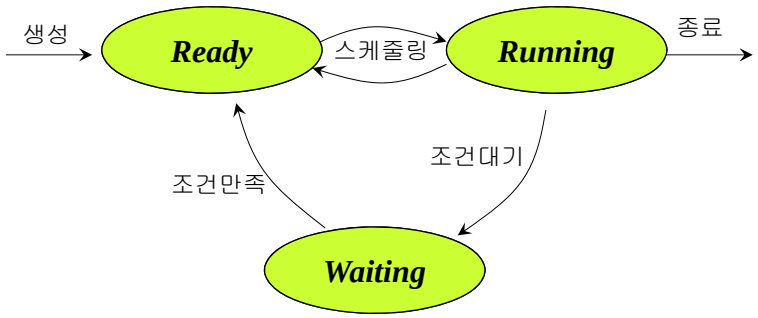
유닉스 (Cygwin) 프로세스 목록

```
~
$ ps
      PID    PPID    PGID     WINPID  TTY  UID    STIME COMMAND
    39604       1   39604      39604  con 1005 16:51:33 /usr/bin/bash
    51136       1   51136      51136  con 1005 11:51:22 /usr/bin/bash
    49480   39604   49480      49492  con 1005 11:51:33 /usr/bin/vim
    47496   51136   47496      32984  con 1005 11:52:58 /usr/bin/ps

김용석@your-4899f23bb4 ~
$
```


프로세스의 일생과 상태천이

- ✓ 프로세스 상태
  - ✓ Running: CPU에 의해 현재 실행중인 상태
  - ✓ Ready: 실행가능 상태로서 스케줄링 과정에서 선택되면 Running 상태로 전환됨
  - ✓ Waiting: 특정 조건이 만족될 때까지 대기하고 있는 상태
    - ✓ 또는 Blocked 상태라고도 함
  - ✓ Suspended: 강제로 실행이 중지된 상태로서 별도로 실행재개 요청이 있어야 실행 가능함
- ✓ 스케줄링
  - ✓ Ready 상태의 프로세스들 중에서 일정한 원칙에 의해 하나를 선택하는 작업
  - ✓ 스케줄러: 스케줄링 작업을 하는 운영체제 부분으로서 보통 함수형태로 구현하고 스케줄링이 필요한 시점마다 이 함수를 호출함
  - ✓ 선점: 스케줄링 결과에 의해 Running 상태의 프로세스를 강제로 다른 프로세스로 변경하여 실행

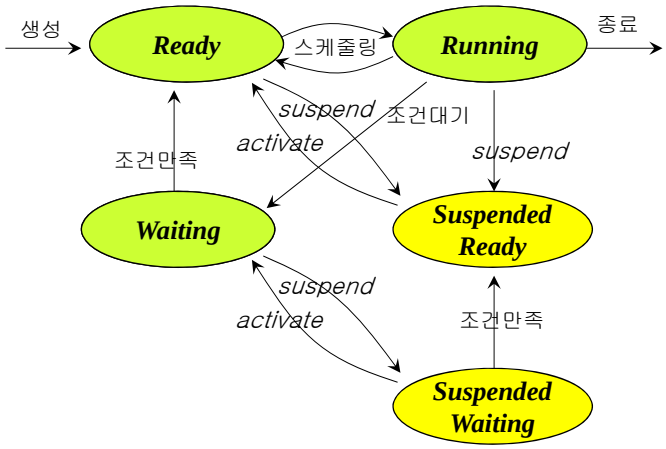
간단한 프로세스 상태천이도



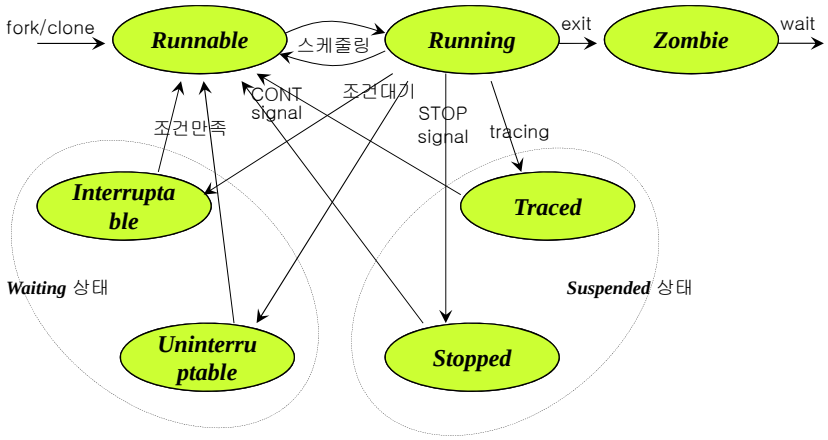
Waiting 상태로 전환되는 상황들?
강제종료 요청시 처리는?

프로세스 상태 천이도

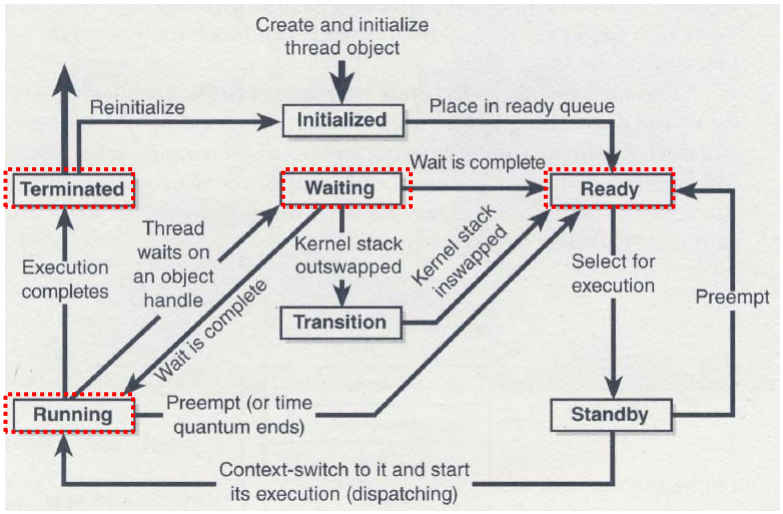
✓ Suspended 상태 포함



리눅스의 상태천이도



Windows NT Thread State



프로세스 제어블록

- ✓ 프로세스 제어블록 (process control block)
  - ✓ 프로세스 별로 정보를 모아서 기록하여 관리하는 자료 구조
  - ✓ 프로세스 생성시 PCB를 하나 할당하여 필요한 정보 기록
  - ✓ 일명 process descriptor, task control block, task structure
- ✓ PCB의 주요 기록 정보
  - ✓ 프로세스 현재 상태
  - ✓ 스케줄링 정보 (우선순위, 실행시간 등)
  - ✓ 프로세스 별 고유번호
  - ✓ 사용자 관련 정보 (사용자 번호, 총 사용시간 등)
  - ✓ 메모리 영역 정보 (코드, 데이터, 스택 영역 주소)
  - ✓ 사용중인 자원 (open 상태의 파일 목록 등)
  - ✓ 문맥교환 과정에서 저장할 정보 (레지스터 값들, 다음 명령어 주소, 현재 스택 탑 주소 등)
  - ✓ 프로세스 종료 코드
  - ✓ 목록으로 관리하기 위한 항목 (next, child, parent 등)
  - ✓ (멀티코어이면, 이 프로세스를 실행하는 core ID도 기록)

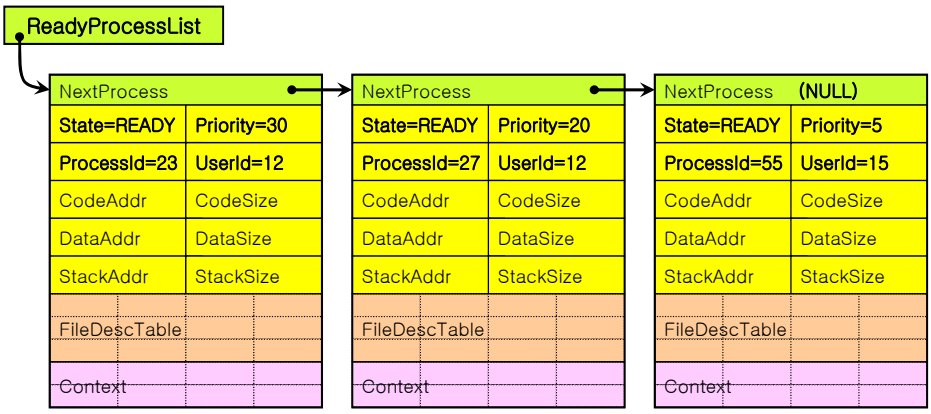
간단한 프로세스 제어블록

NextProcess	
State	Priority
ProcessId	UserId
CodeAddr	CodeSize
DataAddr	DataSize
StackAddr	StackSize
FileDescTable	
Context	

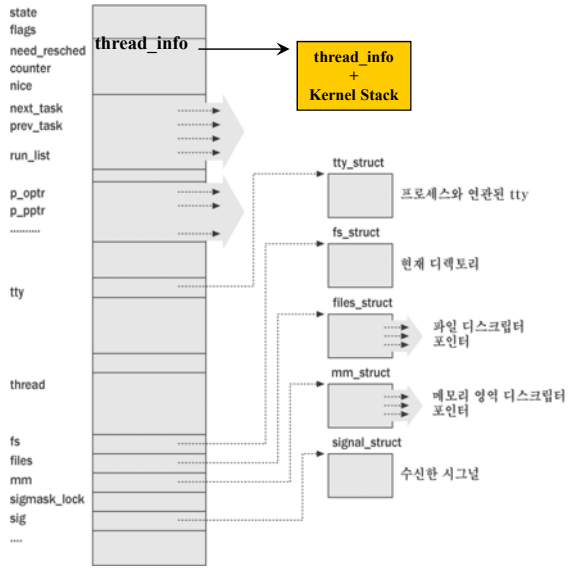
```
struct ProcessControlBlock {
   struct ProcessControlBlock
      *NextProcess;
   int State, Priority;
   int ProcessId, UserId;
   unsigned long CodeAddr,
      DataAddr, StackAddr;
   int CodeSize, DataSize,
      StackSize;
   struct OpenFile
      *FileDescTable[12];
   unsigned long Context[8];
}
```


Ready 프로세스 목록

- ✓ Ready 상태의 프로세스들을 목록으로 관리
- ✓ 스케줄링 과정에서 이 목록을 검사



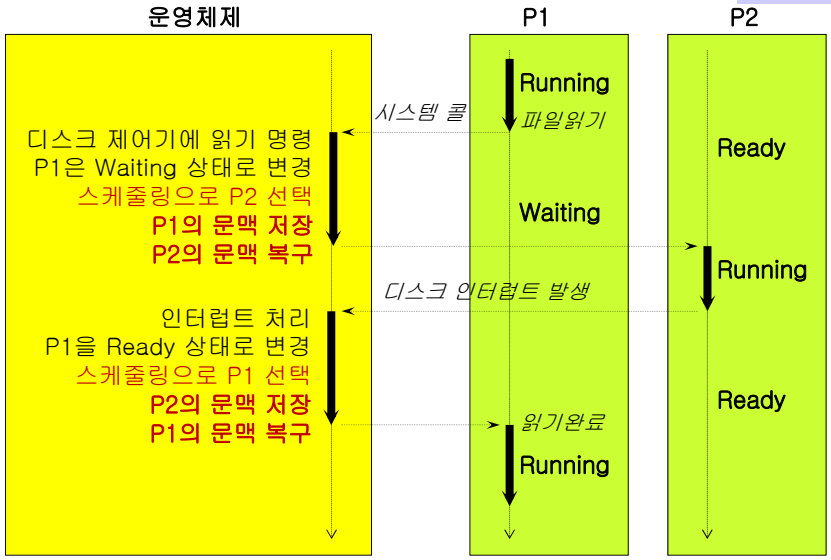
[참고] Linux struct task_struct



문맥과 문맥교환

- ✓ 문맥과 문맥교환 (context switch)
  - ✓ 문맥: 특정 시점의 프로세스의 현재 상태
  - ✓ 문맥교환: 실행할 프로세스를 다른 것으로 변경하는 작업
- ✓ 문맥에 저장할 정보
  - ✓ 나중에 다시 선택되어 실행 재개시 원상 복구해야 할 모든 정보
  - ✓ 현재 실행하던 위치, 현재 스택 탑 위치, CPU 내부의 레지스터 값들
  - ✓ 데이터 영역 및 스택 영역의 현재 내용
  - ✓ 사용자에 대한 정보, 사용중인 메모리 영역 정보, 사용중인 자원들의 정보
- ✓ 문맥교환시 실제 저장할 정보
  - ✓ CPU 내부의 레지스터 값들 (현재 실행위치, 스택 탑 위치 포함)은 필수적으로 프로세스 별 문맥 저장 영역에 기록 (하드웨어 문맥)
- ✓ 문맥교환시 저장하지 않아도 되는 정보
  - ✓ 메모리 내용: 데이터 영역 및 스택 영역은 프로세스 별로 독립적
  - ✓ PCB 내용: PCB에 저장하는 정보들도 프로세스 별로 독립적 (하드웨어 문맥 별도)

프로세스간의 문맥교환 절차



문맥교환 알고리즘 예

✓ 문맥교환 알고리즘 예

```
prev = RunningProcess;      // 실행 중인 프로세스 제어블록
next = Schedule();          // 실행할 프로세스 제어블록 선택
if (next != prev) {         // 문맥교환을 해야 한다면
  메모리 관리 기능을 위한 페이지 테이블 변경 작업;
  if (prev->State == RUNNING)
    prev->State = READY;    // 이전 프로세스를 Ready로
  next->State = RUNNING;    // 선택된 프로세스는 Running으로
  RunningProcess = next;    // 실행 중인 프로세스 변경
  ContextSave(prev->Context);     // 이전 프로세스 문맥저장
  ContextRestore(next->Context); // 실행할 프로세스 문맥복구
}
```

✓ 문맥저장 및 복구 함수
- ✓ 문맥 저장: CPU 내부의 레지스터 값들을 메모리 영역에 기록
- ✓ 문맥 복구: 메모리로부터 저장된 값들을 읽어서 레지스터에 복구

➔ 어셈블리 언어로 작성해야 함

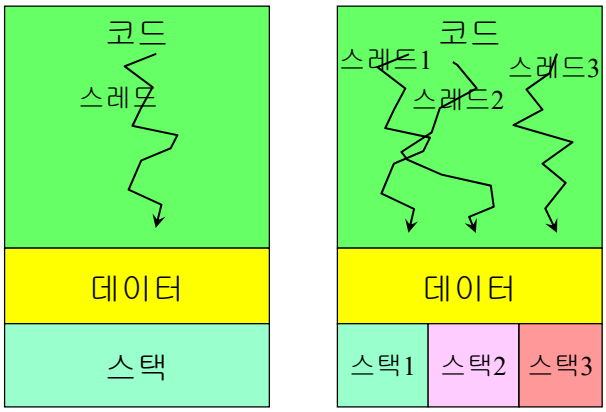
멀티스레드 프로세스

- ✓ 스레드 (thread)
  - ✓ 프로세스를 구성하는 독립된 실행단위
  - ✓ 하나의 프로세스에 부여된 작업을 여러 스레드들이 나누어서 실행
- ✓ 스레드 간의 메모리 공유
  - ✓ 동일한 프로세스에 소속된 스레드 간에는 코드영역 및 데이터 영역을 공유하여 사용
  - ✓ 각 스레드는 독자적으로 코드영역의 함수들을 실행하면서 지역변수들을 사용 ➔ 스택은 스레드 별로 별도로 할당

✓ (참고) 스레드를 경량 프로세스 (light weight process)로, 프로세스를 중량 프로세스 (heavy weight process) 로 구분하기도 함

단일/멀티 스레드 프로세스

✓ 단일스레드 프로세스와 멀티스레드 프로세스



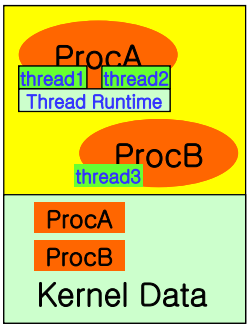
멀티스레드 프로세스의 장단점

✓ 스레드 사용의 장점
- ✓ 스레드 간의 병행 처리로 작업 완료시간 단축
  - ← 입출력 대기시간 동안 다른 스레드 실행 가능
- ✓ 프로그램을 기능모듈 별로 분리하여 작성 가능
  - → 프로그램 개발 용이
- ✓ 비슷한 작업을 동시에 많은 개수를 실행할 때 (웹 서버 등)
  - → 프로그램 개발 용이
- ✓ 멀티코어 CPU일 경우 스레드들을 병렬로 실행가능

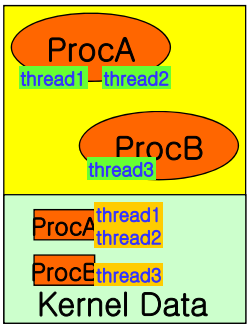
✓ 스레드 사용의 단점
- ✓ 스레드 간의 스케줄링 및 문맥교환 오버헤드 증가
- ✓ 필요한 스택 영역 크기 총량 증가
- ✓ 프로그램 디버깅 작업이 복잡함 (특히 스택 오버플로우 문제에서)

스레드의 구현 방법

- **User Level Thread**
  - Library 함수로 구현
  - OS의 지원이 필요 없음
  - Process 단위로 스케줄링
  - 오버헤드는 낮지만 병행처리에 제한적임



- **Kernel Level Thread**
  - OS에서 Thread들을 구별
  - System Call로 Thread 생성 요청
  - Thread 단위로 스케줄링
  - Multi-processor 장점 활용



프로세스와 스레드의 지향점

- 하나의 프로세스는 고유의 목적을 가진 독립적인 개체
  - 자식 프로세스는 부모 프로세스로부터 부여받은 업무를 수행한 후에 결과 보고
- 하나의 프로세스에 소속된 스레드들은 공동의 목적을 가지고 서로 협조하는 공동 운명체
  - 스레드 하나에 오류가 있으면 동료 스레드 모두 강제종료
- 참고
  - 프로세스 간에는 부모만이 자식종료 대기 가능 (process wait)
  - 동료 스레드 간에는 서로 종료 대기 가능 (thread join)

스레드 간의 공통 정보는?

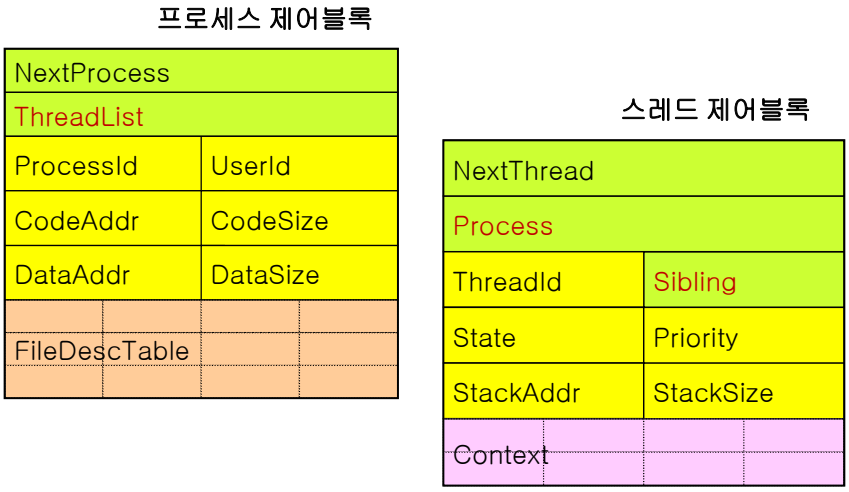
- ✓ PCB에 저장해야 할 정보들 중에서 스레드간에 공통정보와 개별정보는?
- ✓ 착안: 스케줄링은 스레드 단위로 처리함

- ✓ 프로세스 상태 → 스레드 상태
- ✓ 스케줄링 정보 (우선순위, 실행시간 등) ?
- ✓ 프로세스 별 고유번호 ?
- ✓ 사용자 관련 정보 (사용자 번호, 총 사용시간 등) ?
- ✓ 메모리 영역 정보 (코드, 데이터, 스택 영역 주소) ?
- ✓ 사용중인 자원 (open 상태의 파일 목록 등) ?
- ✓ 문맥교환 과정에서 저장할 정보 (레지스터 값들, 다음 명령어 주소, 스택 상단 주소 등) ?
- ✓ 목록으로 관리하기 위한 항목 (next, child, parent 등) ?
- ✓ 종료 코드 ?

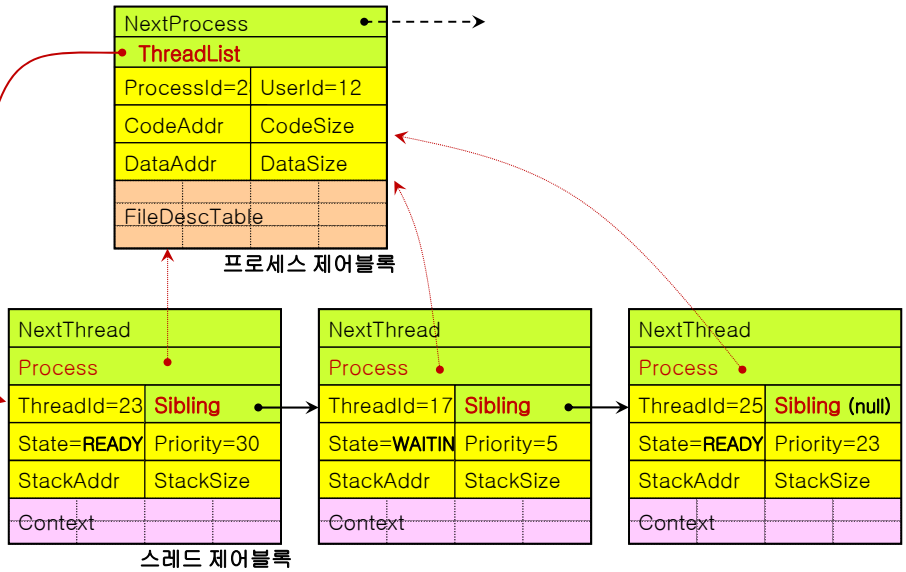
프로세스/스레드 제어블록

- ✓ 프로세스 제어블록 관리 항목
  - ✓ 프로세스별 고유번호
  - ✓ 사용자 관련 정보 (사용자 번호, 총 사용시간 등)
  - ✓ 사용중인 코드 및 데이터 영역 메모리 범위 정보
  - ✓ 사용중인 자원 (open 상태의 파일 목록 등)
  - ✓ 소속 스레드들의 목록
  - ✓ 프로세스 목록을 관리하기 위한 항목
  - ✓ 종료 코드
- ✓ 스레드 제어블록 관리 항목
  - ✓ 스레드 별 고유번호
  - ✓ 스레드 현재 상태
  - ✓ 스케줄링 정보 (우선순위, 실행시간 등)
  - ✓ 사용중인 스택 영역 정보
  - ✓ 문맥교환 과정에서 저장할 정보 (레지스터 값들)
  - ✓ 자신이 소속한 프로세스를 가리키는 항목
  - ✓ 동일한 프로세스에 소속된 동료 스레드들의 목록 관리 항목
  - ✓ 스레드들의 목록을 관리하기 위한 항목

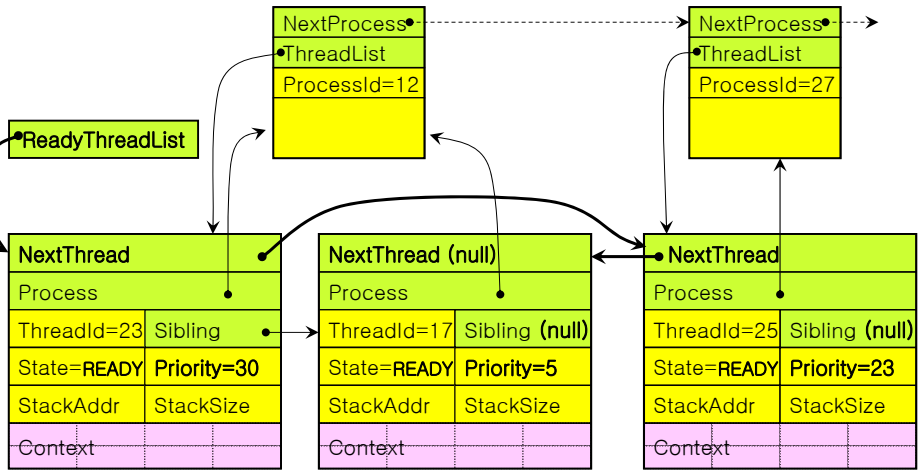
프로세스/스레드 제어블록 예



3개의 스레드를 가진 프로세스 예



Ready 상태의 스레드 목록 예



스레드 간의 문맥교환 알고리즘

```
prev = RunningThread;              // 현재 실행중인 스레드 제어블록
next = Schedule();                 // 실행할 스레드 제어블록 선택
if (next != prev) {                // 문맥교환을 해야 한다면
  if (next->Process != prev->Process) // 서로 다른 프로세스
    메모리 관리 기능을 위한 페이지 테이블 변경작업;
  if (prev->State == RUNNING)
    prev->State = READY;           // 이전 스레드의를 Ready로
  next->State = RUNNING;           // 선택된 스레드는 Running으로
  RunningThread = next;            // 실행 중인 스레드 변경
  ContextSave(prev->Context);      // 이전 스레드의 문맥 저장
  ContextRestore(next->Context); // 실행할 스레드의 문맥 복구
}
```

✓ 서로 다른 프로세스에 소속된 스레드이면 사용 메모리 영역이 다르므로 사용 메모리 영역 변경 작업 필요

스레드의 생성 및 종료

- ✓ 프로세스의 시작시 스레드는 하나
- ✓ 프로그램 상에 스레드 생성 시스템 콜 함수 호출로 필요한 스레드들을 추가 생성
- ✓ 마지막 스레드가 종료되면 프로세스도 종료
- ✓ 프로세스 종료 시스템 콜 함수가 호출되면 소속된 모든 스레드들도 종료
- ✓ 동료 스레드 간에는 상대방이 종료할 때까지 기다리는 시스템 콜 함수(join)도 제공

멀티스레드 프로그램 예 (pico)

```
int userMain(int arg)
{
  threadCreate(20, BodyA, 0);
  threadCreate(20, BodyB, 0);
  threadCreate(20, BodyB, 0);
}
int BodyA(int arg)
{
  int myid = threadSelf();
  int count = 0;
  while(count <= 20) {
    count++;      printf("(A%d:%d)", myid, count);
    threadYield();
  }
}
int BodyB(int arg)
{
  int myid = threadSelf();
  int count = 0;
  while(count <= 20) {
    count++;      printf("(B%d:%d)", myid, count);
    threadYield();
  }
}
```


(참고) 멀티스레드 프로그램 예 (POSIX)

```
int main(int arg)
{
  pthread_create(NULL, NULL, BodyA, NULL);
  pthread_create(NULL, NULL, BodyB, NULL);
  pthread_create(NULL, NULL, BodyB, NULL);
}
int BodyA(void *arg)
{
  pthread_t myid = pthread_self();
  int count = 0;
  while(count <= 20) {
    count++;      printf("(A%d:%d)", myid, count);
    pthread_yield();
  }
}
int BodyB(void *arg)
{
  pthread_t myid = pthread_self();
  int count = 0;
  while(count <= 20) {
    count++;      printf("(B%d:%d)", myid, count);
    pthread_yield();
  }
}
```

(참고) 멀티스레드 프로그램 예 (Java)

```
public class MyProg
{
  public static void main(String[] args) {
    BodyA rA = new BodyA();
    BodyB rB = new BodyB();
    Thread t1 = new Thread(rA);
    Thread t2 = new Thread(rB);
    Thread t3 = new Thread(rB);
    t1.start(); t2.start(); t3.start();
  }
}
public class BodyA implements Runnable
{
  public void run(){
    int myid = this.getId();
    int count = 0;
    while(count <= 20) {
      count++;
      System.out.print("(A" + myid + ":" + count + ")");
      Thread.yield();
    }
  }
}
public class BodyB implements Runnable { . . . }
```


실행결과 출력

- ✓ 실행 결과 출력 (하나의 창에 표시)

```
(A2:1)(B3:1)(B4:1)(A2:2)
(B3:2)(B4:2)(A2:3)(B3:3)
(B4:3)(A2:4)(B3:4)(B4:4)
(A2:5)(B3:5)(B4:5)(A2:6)
(B3:6)(B4:6)
 ......
```

- ✓ 스레드별로 별도 창에 출력한다면

스레드 2

```
(A2:1)(A2:2)
(A2:3)(A2:4)
(A2:5)(A2:6)
 ......
```

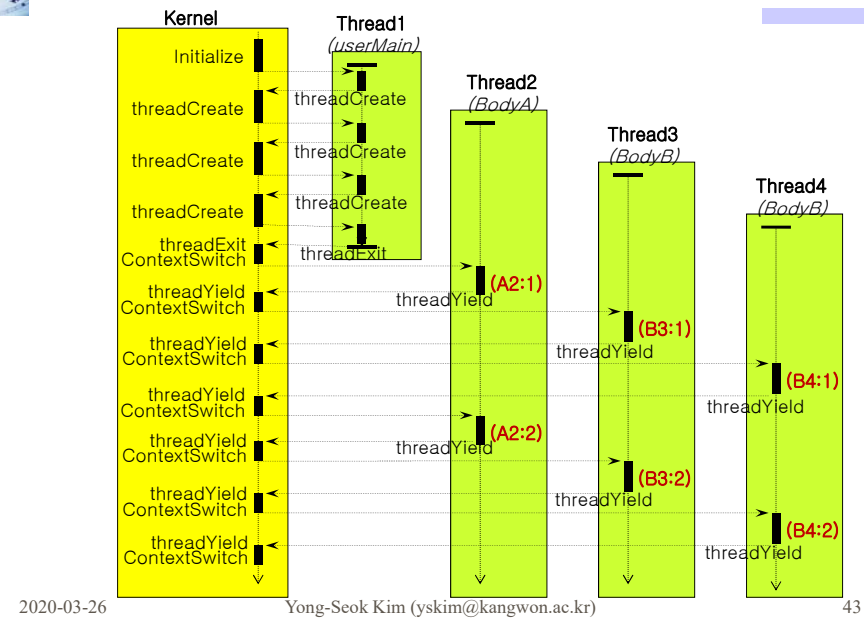
스레드 3

```
(B3:1)(B3:2)
(B3:3)(B3:4)
(B3:5)(B3:6)
 ......
```

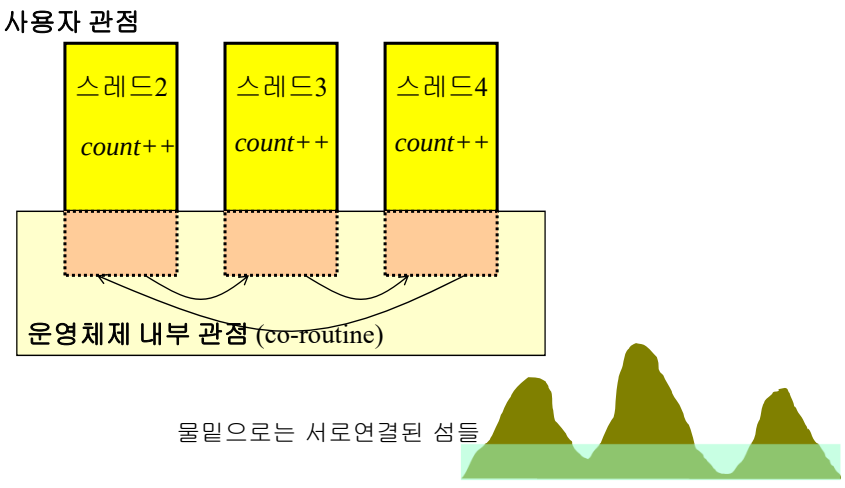
스레드 4

```
(B4:1)(B4:2)
(B4:3)(B4:4)
(B4:5)(B4:6)
 ......
```

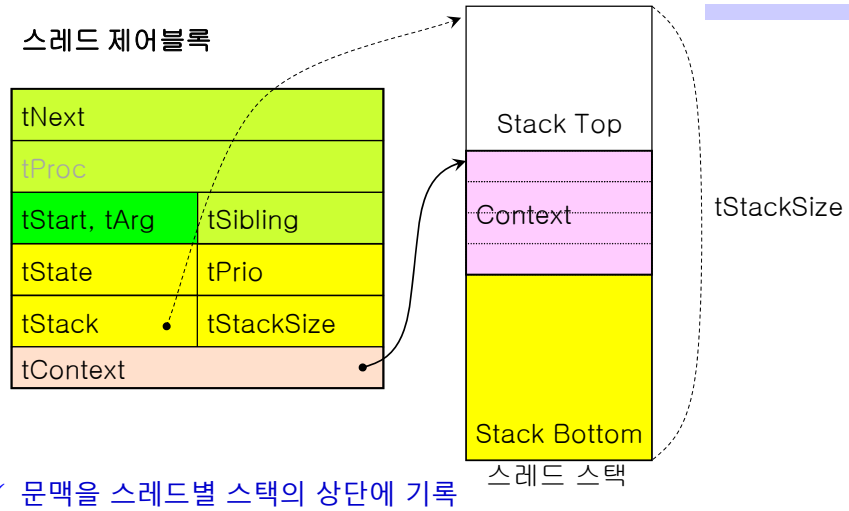
시간에 따라 CPU가 실행하는 내용



스레드별 독립적인 실행 모습



picoKernel의 문맥교환



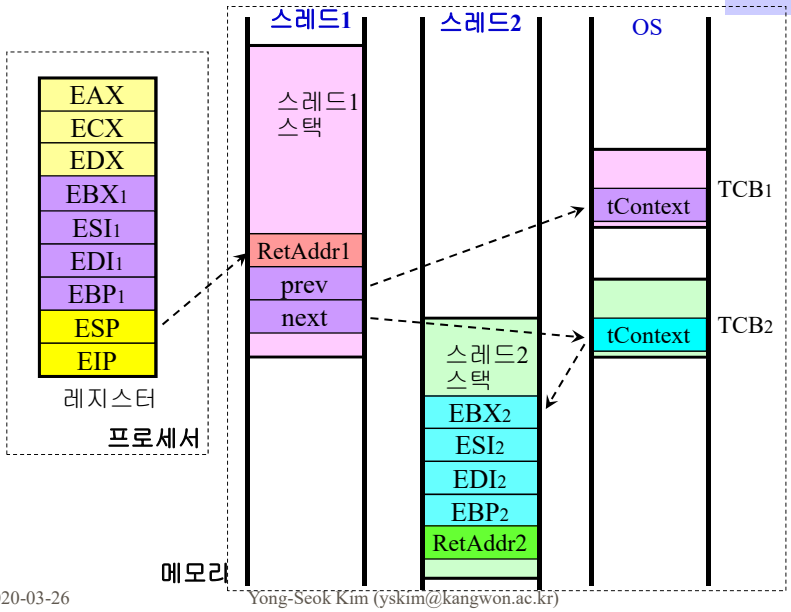
- ✓ 문맥을 스레드별 스택의 상단에 기록
- ✓ 저장된 문맥 영역의 주소는 스레드 제어블록의 tContext 에 표시
- ✓ contextSwitch(&(prev->tContext), &(next->tContext));

contextSwitch 함수

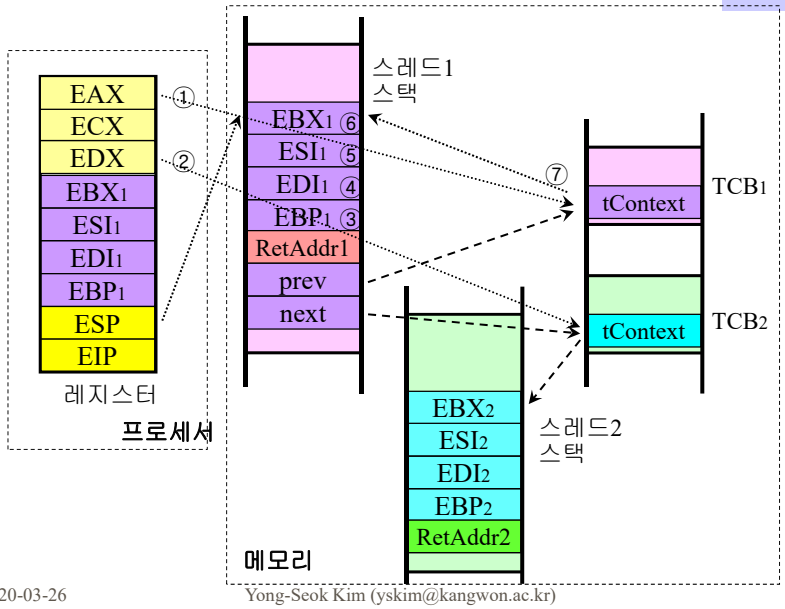
이전 스레드의 문맥 저장과 다음 스레드의 문맥복구를 하나의 함수로 처리

```
# contextSwitch(prev, next)
# ESP --> | RetAddr  |  # parameters on the stack
#         |    prev  |
#         |    next  |
contextSwitch:
    movl 4(%esp),%eax    # (1) get prev context addr
    movl 8(%esp),%edx    # (2) get next context addr

    # save the context of the prev thread
    pushl %ebp           # (3) save EBP
    pushl %edi           # (4) save EDI
    pushl %esi           # (5) save ESI
    pushl %ebx           # (6) save EBX
    movl %esp,(%eax)     # (7) save ESP

    # load the context of the next thread
    movl (%edx),%esp     # (8) load ESP
    popl %ebx            # (9) load EBX
    popl %esi            # (10) load ESI
    popl %edi            # (11) load EDI
    popl %ebp            # (12) load EBP
    ret                  # (13) return
```

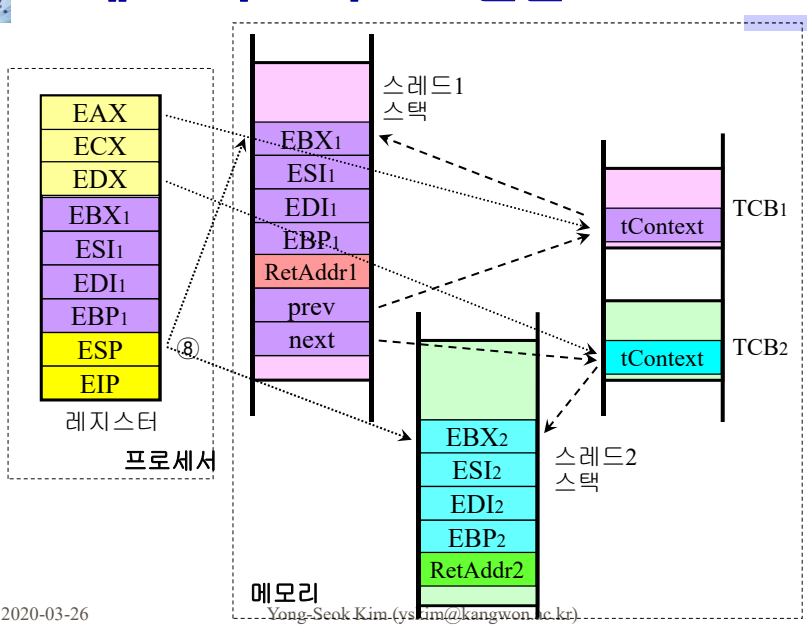
문맥교환함수의 시작시점



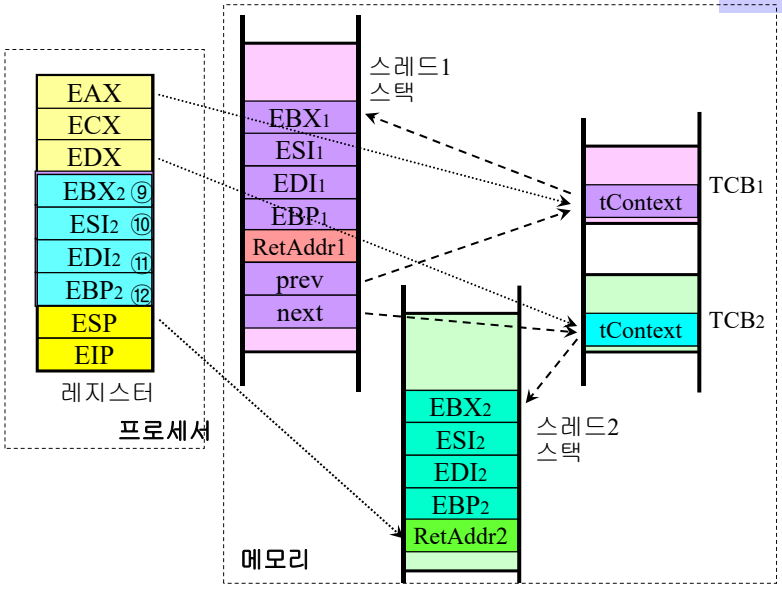
스레드1의 문맥 저장후



스레드2의 스택으로 전환



스레드2의 문맥 복구후

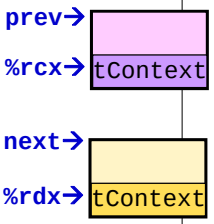


(참고) Gcc 컴파일러 x86 64비트 버전

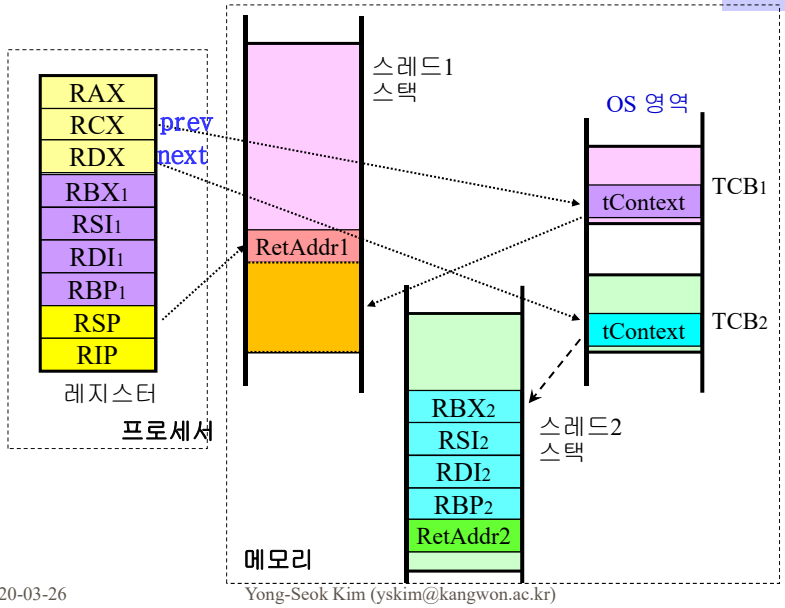
- ✓ 레지스터와 명령어 표기
  - ✓ 64비트 레지스터: RAX, RBX, …, RSP, RIP, R8, R9, … R15
  - ✓ 64비트 명령어: movq %rax, %rcx
  - ✓ *(참고) R8의 일부분 명칭: 하위 32bit는 R8D (double word), 16bit는 R8W, 8bit는 R8L*
- ✓ 변수 크기
  - ✓ char 8bit, short 16bit, int 32bit, long 64bit
  - ✓ 주소: 64bit
- ✓ 파라미터 전달 방법 (Cygwin gcc)
  - ✓ 4개까지는 차례대로 %rcx, %rdx, %r8, %r9를 활용
  - ✓ 5개 부터는 스택에 저장하여 전달
- ✓ 파라미터 전달 방법 (Linux gcc)
  - ✓ 6개까지는 차례대로 %rdi, %rsi, %rdx, %rcx, %r8, %r9를 활용
  - ✓ 7개 부터는 스택에 저장하여 전달

(참고) contextSwitch x86 64비트 버전

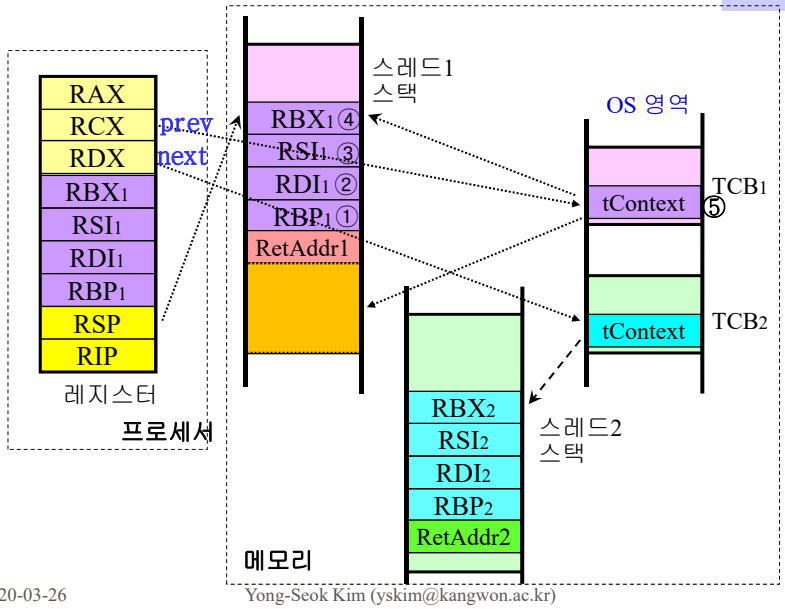
```
# call: contextSwitch(&(prev->tContext), &(next->tContext));
# contextSwitch(prevc, nextc)   # Cygwin gcc
# RSP --> | RetAddr  |
# parameter passing: %rcx = prevc, %rdx = nextc
contextSwitch:
       # save the context of the prev thread
        pushq %rbp              # (1) save RBP
        pushq %rdi              # (2) save RDI
        pushq %rsi              # (3) save RSI
        pushq %rbx              # (4) save RBX
        movq %rsp,(%rcx)        # (5) save RSP

       # load the context of the next thread
        movq (%rdx),%rsp        # (6) load RSP
        popq %rbx               # (7) load RBX
        popq %rsi               # (8) load RSI
        popq %rdi               # (9) load RDI
        popq %rbp               # (10) load RBP

        ret                     # return from this function
```



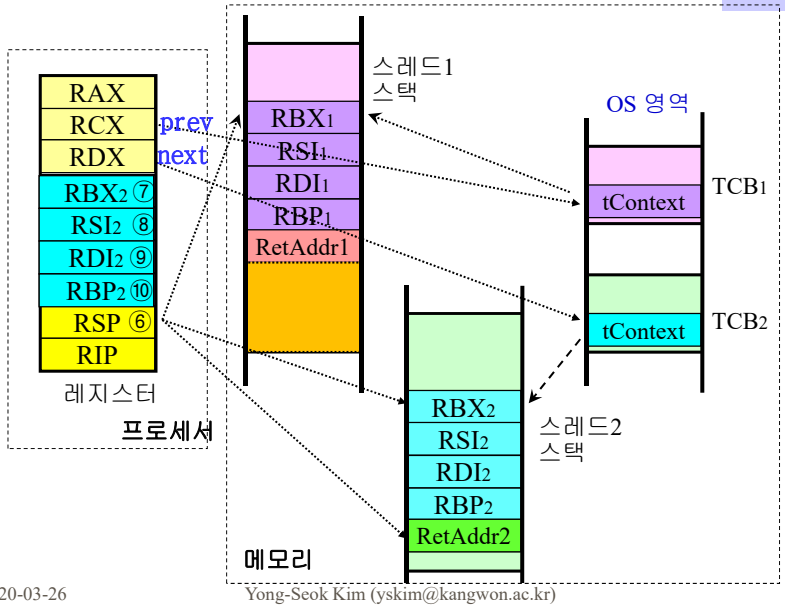
문맥교환함수의 시작시점



스레드1의 문맥 저장과정



스레드2의 문맥 복구과정



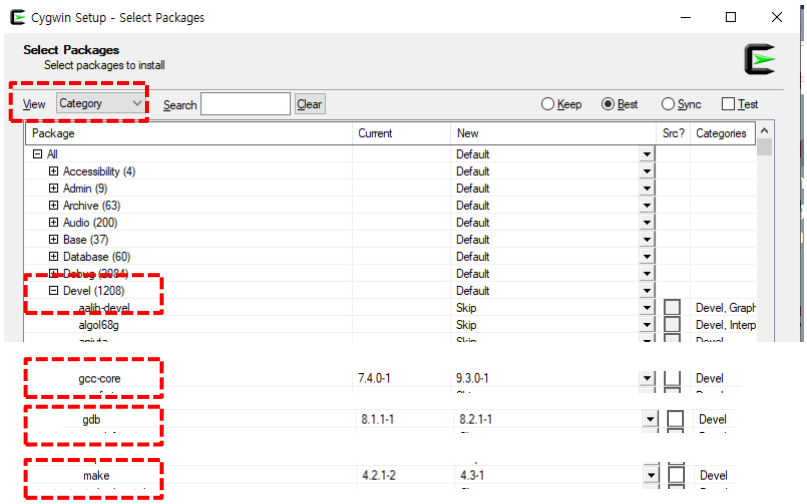
picoKernel의 실행

- ✓ 실행 환경
  - ✓ Linux
  - ✓ Cygwin on Windows: www.cygwin.com
- ✓ picoKernel의 실행
  - ✓ pico3.2.2.tar 파일 다운로드: 과목 게시판
  - ✓ tar file 풀기
  - ✓ context64-cygwin.s를 context.s로 복사
  - ✓ make all 또는 make fig3-16
  - ✓ ./fig3-16

- pico3.2
  - kernel
    - kernel.c
    - kernel2.c
    - context32-linux.s
    - context32-cygwin.s
    - context64-linux.s
    - context64-cygwin.s
    - context.s
  - makefile
  - fig3-16.c
  - fig9-1.c
  - …
  - context.o
  - kernel.o
  - kernel2.o

www.cygwin.com

Setup-x86_64.exe 실행화면

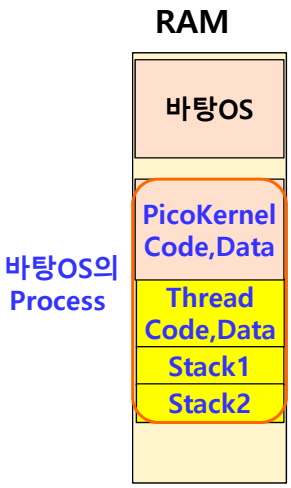


Cygwin 설치

- ✓ www.cygwin.com
  - ✓ 왼쪽 상단의 'Install Cygwin' 클릭 (64bit 또는 32bit 버전 설치)
  - ✓ Setup-x86_64.exe 또는 setup-x86.exe (32비트 버전)를 받아서 실행
  - ✓ 설치과정에서 Devel/gcc-core, gdb, make 포함 필수
- ✓ 강의 게시판
  - ✓ picoKernel 3.2.2 다운로드: pico3.2.2.tar
  - ✓ Windows 에서 C:₩cygwin₩home₩(Administrator) 폴더에 복사
- ✓ Cygwin 실행 후
  - ✓ tar xvf pico3.2.2.tar 또는 unzip pico3.2.2.zip ← 압축 풀기
  - ✓ cd pico3.2.2
  - ✓ cp kernel/context64-cygwin.s kernel/context.s
  - ✓ make all
  - ✓ ./fig3-16
- ✓ Myprog.c 편집 및 실행
  - ✓ gcc –o myprog myprog.c kernel.o context.o
  - ✓ ./myprog

picoKernel의 스레드 실행

- ✓ 바탕 OS의 프로세스로 실행
- ✓ int 명령 사용 안함
- ✓ 하드웨어 인터럽트 없음
- ✓ Suspended 상태 없음



```
// definition of thread entry structure
enum threadState {STATE_READY, STATE_WAITING};

typedef struct threadEnt {
    struct threadEnt *tNext;       // point to next thread entry

    enum threadState tState;       // thread state
    int tPrio;                     // thread priority

    int (*tStart)(int);            // start function address
    int tArg;                      // argument of tStart

    char *tStack;                  // stack base address
    int tStackSize;                // stack size in bytes

    unsigned long tContext;        // address of context area

#ifdef PROCESS
    struct procEnt *tProc;         // process control block
    struct threadEnt *tSibling;    // list of sibling threads
#endif

} threadEntT;

#define       NUM_THREAD    30     // maximum number of threads

// declaration of the thread table
threadEntT threadTab[NUM_THREAD];

// declarations of major kernel variables
threadEntT *threadRunning;         // the running thread
threadEntT *threadReadyList;       // ready thread list
threadEntT *threadFreeList;        // free thread entry list
```

```
int threadCreate(
    int prio,               // thread priority
    int (*start)(int),      // start function address
    int arg                 // thread argument
) {
    threadEntT *thread;

    // allocate a thread entry from free list
    thread = threadFreeList;
    threadFreeList = threadFreeList->tNext;  // delete from free list

    thread->tStart = start;         // thread start function address
    thread->tArg = arg;             // thread argument
    thread->tPrio = prio;           // thread priority

    // set initial context (context area is on the bottom of stack)
    thread->tContext = (unsigned long)(thread->tStack)
              + thread->tStackSize - 24;
    contextSet(thread->tContext, (unsigned long)(threadStartup));

#ifdef PROCESS
    // insert into the thread list of the same process
    thread->tProc = threadRunning->tProc;
    thread->tSibling = threadRunning->tSibling;
    threadRunning->tSibling = thread;
#endif

    // insert into the ready list
    thread->tState = STATE_READY;
    schedInsertReady(thread);

    // reschedule to run the highest priority thread
    schedRunHighest();

    return thread - threadTab;      // return the thread ID
}
```


```
void threadStartup(void)
{
    // call the actual start function given on thread creation
    (*(threadRunning->tStart))(threadRunning->tArg);

    // now, return from the start function and want to exit

    threadExit();    // never return from this function
}
```

```
int threadExit(void)
{
#ifdef PROCESS
    // delete from the thread list of the process
    procDeleteThread(threadRunning);
#endif

    // remove the running thread from ready list
    schedDeleteRunning();

    // insert it into free list
    threadRunning->tNext = threadFreeList;
    threadFreeList = threadRunning;

    // reschedule to run the highest priority thread
    schedRunHighest();      // never return from this function
}
```

threadYield() 와 threadSelf()

```
int threadYield(void)
{
    // remove the running thread from ready list
    schedDeleteRunning();

    // reinsert it into ready list
    schedInsertReady(threadRunning);

    // reschedule to run the highest priority thread
    schedRunHighest();

    return 0;
}

int threadSelf(void)
{
    return threadRunning - threadTab;
}
```


커널 초기화

```
int main()
{
    unsigned long dummy;    // dummy context address

    kernelInit();           // initialize data structures and devices
    bootMessage();          // display booting message

    // create system threads
    threadCreate(0, nullMain, 0);  // the null thread (priority=0)
    threadCreate(21, userMain, 0); // the first user thread

    // now change to multi-tasking mode
    threadRunning = threadReadyList;
    contextSwitch(&dummy, &threadRunning->tContext);
    // never come here
}

int nullMain(int arg)
{
    while (1)               // loop forever
       threadYield();       // yield to others
}
```

ISR 등록

✓ (참고) picoKernel에는 없는 가상적인 코드임

```
-----------------------------------------------------------------------
// interrupt vector number of a pseudo CPU
#define VEC_ADDR_ERROR      0x01   // address error interrupt
#define VEC_TIMER           0x02   // timer interrupt
#define VEC_DISK            0x10   // disk interrupt
#define VEC_INT80           0x80   // int 0x80 instruction interrupt

int isrInit()
{
    unsigned long *tab;

    tab = getIsrTableAddr();       // get ISR table addr
    tab[VEC_ADDR_ERROR] = (long)addrErrorIsr;   // (프로그램 6.2)
    tab[VEC_TIMER] = (long)timerIsr;            // (프로그램 4.1)
    tab[VEC_DISK] = (long)diskIsr;              // (프로그램 8.1)
    tab[VEC_INT80] = (long)int80Isr;            // (그림 2.18)
}
-----------------------------------------------------------------------
```