파이썬 – HW9

임베디드스쿨1기

Lv1과정

2020. 10. 07

강경수

1. Critical Section

▣ Critical Section

2020.10.05

1. 크리티컬 섹션(Critical Section)이란?
- 서로 다른 두 프로세스, 혹은 스레드 등의 처리 단위가 같이 접근해서는 안 되는 공유 영역을 뜻한다.

2. 크리티컬 섹션에 두 개 이상의 프로세스 혹은 스레드가 접근 하는 경우
- PROCESS 관점에서 OS Scheduling에 의해 권한이 중간에 넘어감.

→ EX) mov, sub, add 순서로 진행되는코드가 있다고 가정할시 move명령어 실행후 스케쥴링에 의하여 다른 TASK로 넘어가버리면 크리티컬 섹션에 서로다른 TASK가 접근 할 수 있게 된다.
- 인터럽트에 의해서

1. Critical Section

3. 이런 문제를 방지하기 위한 방법
① 프로세서 간의 간섭방지 세마포어
② 스레드간의 간섭 방지 뮤텍스
③ Context Switching : 진행중이던 TASK의 레지스터값들을 저장하고 다시 해당 TASK호출시 이전상태 복구
④ Spin Lock : Context Swithcing을 하지 않고 loop를 돌며 lock이 풀릴때까지 기다리는 방법(Context Switching을 하지 않아 CPU효율을 높일 수 있음 여기에 loop를 일정시간 돌아도 lock이 풀리지 않으면 잠시 sleep 하는 back off 알고리즘을 사용 할 수 있다.

※ 세마포어와 뮤텍스의 차이점 : 뮤텍스 객체를 두 스레드가 같이 사용 할 수 없음
뮤텍스 : 뮤텍스 객체를 두 스레드가 같이 사용 할 수 없다.
세마포어 : 리소스 상태를 나타내는 카운터
1) Semaphore는 Mutex가 될 수 있지만 Mutex는 Semaphore가 될 수 없다.
2) Semaphore는 소유할 수 없는 반면, Mutex는 소유가 가능하며 소유주가 이에 대한 책임을 진다.
3) Mutex의 경우 Mutex를 소유하고 있는 쓰레드가 이 Mutex를 해제할 수 있다.

2. Thread

▣ THREAD란 무엇인가

2020.10.06 KKS

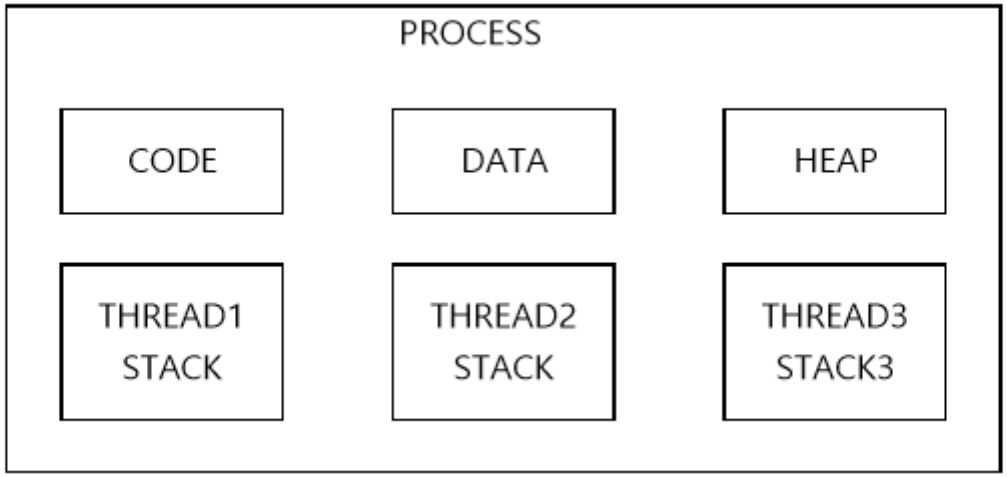
1. Process VS Thread

1)Process : 일반적으로 일컫는 프로그램 이는 내가 C로짠 구구단 프로그램도 해당되고 포토샵도 해당된다.

2) Thread : 한 Process 내부에서 Heap, Data, Code는 공유하며 Stack은 각각 할당받음 프로세스 내에서 실행되는 동작의 흐름

※ Processor : 프로세서는 일반적으로 CPU,MCU등을 포괄정으로 말함.

Processor 하고 Process 하고 헛갈리지마!!



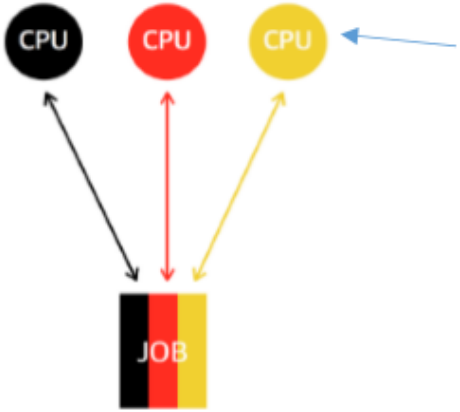
2. Thread

2. Multi Processing VS Multi Thread VS Multi Tasking

1) Multi Processing : 프로그램을 PROCESS들이 나누어 가져 병렬로 처리

장점 : 비용절약(데이터를 공유 할 수 있으므로)

시스템 안정도 증가(프로세서 하나 문제 생겨도 다른것들이 백업)

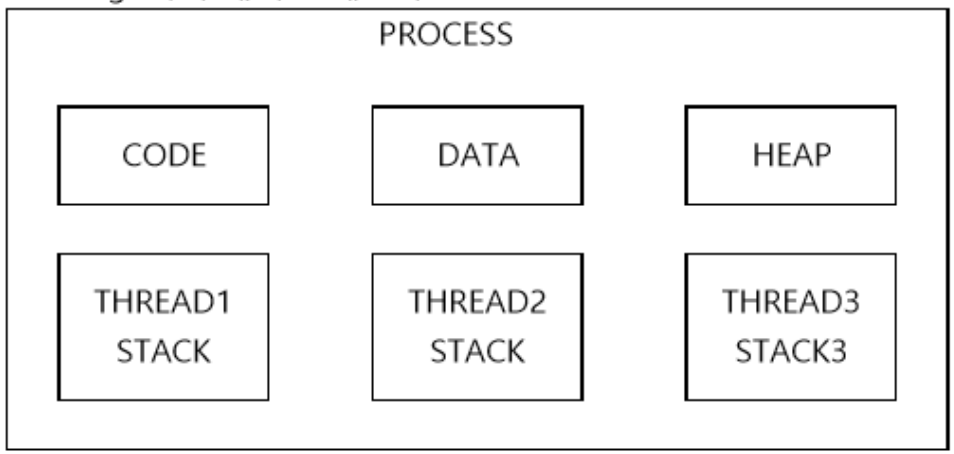


CPU가 아닌 PROCESS

3) Multi Tasking

- Task란 운영체제에서 처리하는 작업의 단위 또는 정해진 일을 수행하기 위한 명령어 집합을 뜻함 (process 보다 확장된 개념), 멀티 태스킹은 task를 OS의 스케쥴링에 의해 task를 번갈아가며 수행하는 것을 의미한다. 여러개의 task를 자주 번갈아가며 수행하다보니 사용자는 동시에 여러 task가 수행되고 있다고 느끼게 된다.

2) Multi Threading: 여러 개의 스레드가 PROCESS를 실행



- DATA공유가 간단함.(멀티프로세싱은 힙 스택이 따로 할당 됨.)
- 하나의 프로그램(프로세스)
- 멀티 스레딩은 멀티 태스킹 보다 데이터 공유가 용이함

3. Nested Interrupt

▣ Nested Interrupt

2020.10.06 KKS

1. Nested Interrupt란 무엇인가

- Interrupt 가 발생하고 그에 대응되는 ISR을 실행중에 다른 인터럽트가 발생되는 경우 새로 발생된 ISR을 처리하는 경우
- Atmega328p를 참조하여 보면 아래와 같이 Nested Interrupt가 사용 가능하다.

> When an interrupt occurs, the global interrupt enable I-bit is cleared and all interrupts are disabled. The user software can write logic one to the I-bit to enable nested interrupts. All enabled interrupts can then interrupt the current interrupt routine. The I-bit is automatically set when a return from interrupt instruction – RETI – is executed.

- 데이터 시트에 의하면 I 비트를 클리어 해줌으로서 자동으로 Nested interrupt를 방지한다. 하지만 이를 인위적으로 0으로 만들어서 Nested Interrupt를 사용하게 할 수 있다.

Q. 만약 Nested interrupt를 사용목적으로 i비트를 다시 set해줄때, 이때 set해주기전 interrupt가 발생하면? → Data loss가 아닐까? 어셈레벨에서 I bit를 clear하는것을 삭제하는 ISR을 직접 만드는것이 방법이 되지 않을까?

```
in      r16, SREG      ; store SREG value
cli                    ; disable interrupts during timed sequence
sbi     EECR, EEMPE    ; start EEPROM write
sbi     EECR, EEPE
out     SREG, r16      ; restore SREG value (I-bit)
```

- Atmel에서 제공하는 ISR루틴을 어셈레벨로 분석하면 위와 같다.
- SREG in,out은 I-bit를 저장하고 다시 set하는 과정(cli가 차단한 인터럽트를 허용함)
- 여기서 cli를 삭제해 버리면 최적화된 Nested interrupt 사용이 가능할듯 싶다.

3. Nested Interrupt

- Atmel에서 제공하는 ISR루틴을 어셈레벨로 분석하면 위와 같다.
- SREG in,out은 I-bit를 저장하고 다시 set하는 과정(cli가 차단한 인터럽트를 허용함)
- 여기서 cli를 삭제해 버리면 최적화된 Nested interrupt 사용이 가능할듯 싶다.

2. 인터럽트 후반부 처리 기법(Bottom half)
 - 빨리 실행할 인터럽트 코드 : 인터럽트 핸들러 및 인터럽트 컨택스트
 - 빨리 실행하지 않아도 되는 인터럽트 코드 : 인터럽트 후반부 기법
- 후반부 기법의 대략적인 컨셉은 아래와 같다.
  ISR루틴을 바로 실행하지 않고 인터럽트가 발생됨을 기억해 뒀다가, 현재 인터럽트 완료후
  나중에 처리해 준다.

※ LINUX에서 BOTTOM HALF 처리해주는 방법
- IRQ THREAD
- Soft IRQ →Linux os level
- 태스크릿 공부가 더 필요함..
- 워크큐

※ RTOS에서 BOTTOM HALF 처리해주는 방법
- xTimerPendFunctionCallFromISR()