실시간 개념 및 운영체제 소개

경희대학교 컴퓨터공학과

조 진 성

주요내용

- 주요내용
  - ✓ 실시간 개념
  - ✓ 실시간 운영체제 이해
  - ✓ 실시간 운영체제 특성
  - ✓ 실시간 운영체제 비교

RTOS의 이해

- RTOS의 Real-Time 단어 때문에, 모든 작업을 실시간으로 처리 할 수 있는 환경을 제공하는 매우 빠른 운영체제라고 하는 오해 존재
- Real-Time이란 임의의 정보가 시스템에 입력 되었을 때 주어진 시간 안에 작업이 완료되어 결과가 주어져야 하는 것을 의미
- RTOS는 주어진 작업을 정해진 시간 안에 수행 할 수 있는 환경 제공
- 예측 가능하고 일정한 응답 시간을 요구하는 응용 프로그램의 지원을 위한 운영체제
- General purpose system에 탑재되는 OS는 H/W 자원을 얼마나 공평하게 분배할 것인지에 주력한 반면 RTOS 에서는 H/W 자원을 좀 낭비하더라고 작업의 시간 제한을 맞추는데 주력
- 공평성의 개념보다는 우선 순위가 높은 task가 많은 시간동안 수행

Hard Real-Time vs Soft Real-Time

- Hard Real-Time
  - ✓ Real-Time System 중에는 어떤 작업을 일정 시간 안에 반드시 처리해야 하며, 그 시간이 지난 후의 결과 값은 정확해도 소용이 없는 경우
  - ✓ Hard Real-Time System
    - Ex) 군사장비, 비행기
- Soft Real-Time
  - ✓ 어떤 시간 안에 처리하면 좋지만 그렇지 못한 경우에 그 시간에서 약간 경과한 후의 값도 인정하는 경우
  - ✓ Soft Real-Time System
    - Ex) cellular phone, router

RTOS - Multitasking

- Multitasking
  - ✓ Embedded system에서의 task는 하나의 문제(목적)를 풀려고 하나의 큰 응용 프로그램을 논리적으로 나눈 개념
  - ✓ 목적을 수행하기 위해서 여러 기능들이 동시에 수행될 필요가 있고 이를 순차적으로 프로그램 하기 어렵기 때문에, 기능 블록의 모듈화 및 CPU의 효율적인 사용이라는 목적에서 Multitasking이라는 개념이 발생
  - ✓ Ex) ADSL Router
    - PPP(point-to-point) Task
    - IP(Internet Protocol) Task
    - UDP(User Datagram Protocol) Task
    - TCP(Transmission Control Protocol) Task
    - RIP(Routing Information Protocol) Task
    - ATM(Asynchronous Transfer Mode) Task

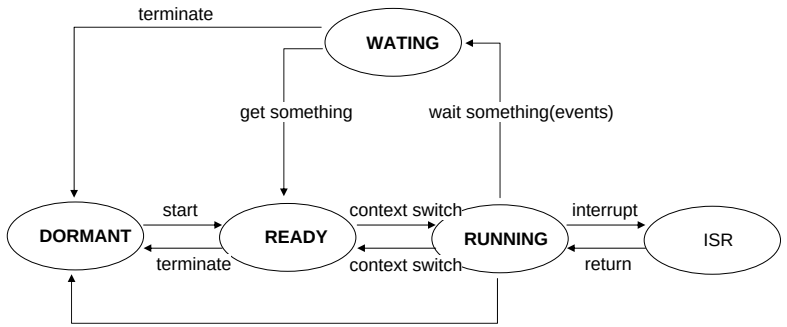
RTOS - Task

- Task priority
  - ✓ Task priority는 이름 그대로 각 task들의 우선순위를 나타내는 것으로 priority가 높은 task가 우선적으로 CPU를 점유하여 실행
  - ✓ Preemptive kernel
- Task stack
  - ✓ Task는 우리가 일반적으로 생각하는 프로그램(함수들의 모임)
  - ✓ Task마다 메모리에 독립된 stack 영역 가지게 되고, 다른 task들은 이 영역을 액세스 할 수 없다
  - ✓ 함수 안에 선언한 지역변수, 함수의 argument, 함수 수행 후 돌아갈 return 어드레스 등이 stack에 저장
  - ✓ 어떤 프로그램이 수행될 때 함수를 호출할 것이며 그 과정에서 stack이 쓰이게 됨

RTOS – Task

- Task status
  - ✓ DORMANT
    - Memory에 존재하나 아직 실행할 수는 없는 상태
    - Task를 생성하면 본 상태로 되지만 어떤 RTOS들은 이 상태가 존재하지 않고 생성시에 바로 READY 상태가 되는 것도 있음
  - ✓ RUNNING
    - CPU를 점유하고 있는 상태
    - Task의 코드가 동작하고 있는 상태
  - ✓ READY
    - 현재 CPU를 사용하고 있는 task보다 priority가 낮아서 CPU 사용의 기회를 기다리고 있는 상태
  - ✓ WATING
    - 어떤 이벤트를 기다리고 있는 상태
    - 만약 기다리고 있던 이벤트가 발생하면 READY 상태로 되어 CPU 사용기회를 기다리게 됨

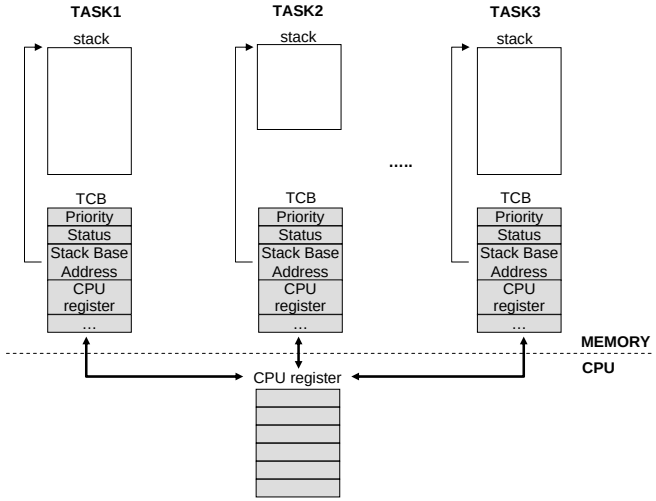
Task states Diagram



Task state 변화 예

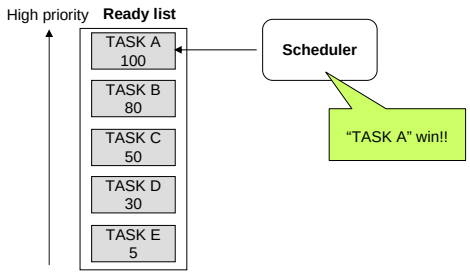
- High priority task A는 serial로부터 데이터를 수신하는 일을 하고 Low priority task B는 주기적으로 LED를 ON/OFF 하는 일을 한다고 가정하면 2개의 task는 다음과 같은 상태변화가 발생
  - ✓ Task A가 task B보다 priority가 높기 때문에 RUNNING 상태가 되고 task B는 CPU 사용 기회를 기다리는 READY상태가 된다.
  - ✓ Task A는 serial로부터 데이터가 수신되지 않았기 때문에 이를 기다리기 위해서 WATING 상태가 되고 Scheduler에 의해서(Context Switch) task B가 RUNNING 상태가 된다. 이제 task B는 루프를 돌면서 주기적으로 LED를 ON/OFF 한다.
  - ✓ Serial로부터 데이터가 수신되면 인터럽트가 발생하고 이에 의해서 ISR이 수행된다. ISR에서는 인터럽트에 대한 처리를 수행하고 Scheduler를 호출한다.
  - ✓ Scheduler에 의해서(Context Switch) ISR 종료와 동시에 WATING 상태에 있던 task A가 RUNNING상태가 되고 task B는 READY 상태로 된다.

Multiple Task Diagram



Scheduler (Dispatcher)

- Scheduler는 READY상태에 있는 여러 개의 task중에 다음에 어떤 task를 수행 시킬 것인지를 결정
- 일반적으로 가장 높은 priority의 task를 수행시키는 "priority-based scheduling" 방법을 사용



Priority based Scheduler

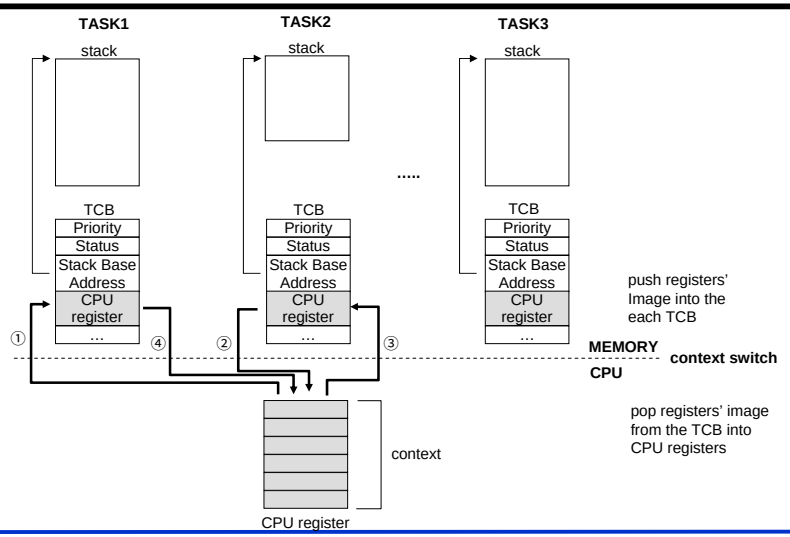
Context Switch (Task Switch)

- 다음 문장들은 모두 동일한 의미
  - ✓ Task가 RUNNING 상태에 있다.
  - ✓ Task가 CPU를 사용(점유)하고 있다.
  - ✓ Task가 CPU의 레지스터를 사용하고 있다.
- Context Switch
  - ✓ Scheduler에 의해서 새로이 RUNNING 상태가 될 task가 결정되면 현재 RUNNING 상태의 task가 사용하던 Context를 메모리 특정 영역에 저장한 후 새로이 수행될 task의 Context를 TCB 또는 스택에서 CPU의 레지스터 영역으로 복사하여 새로운 task가 수행되도록 하는 작업을 Context Switch라 한다.
  - ✓ Task가 사용하는 CPU 레지스터들의 값을 Context라고 한다.
  - ✓ Context는 Task가 유지해야 하는 정보들의 모임으로서 위에서 설명한 것 보다 좀더 포괄적인 의미를 가지고 있다.

Context switch 사례

- 현재 task1이 수행되고 있고 Scheduler에 의해서 READY 상태에 있던 task2가 수행되어야 할 경우에 첫번째 Context switch (①, ②)가 발생한다.
  - ① task1의 Context를 task1의 TCB에 저장
  - ② task2의 TCB에 저장되어 있던 Context를 CPU 레지스터로 복사. 이 시점부터 task2의 코드가 수행
- Scheduler에 의해서 task1을 다시 RUNNING상태로 만든다면 두 번째 Context Switch가 발생(③, ④)
  - ③ task2가 사용하던 CPU 레지스터 값들(Context)을 task2의 TCB에 저장
  - ④ task1의 TCB에 저장되어 있던 Context를 CPU 레지스터 값으로 복사

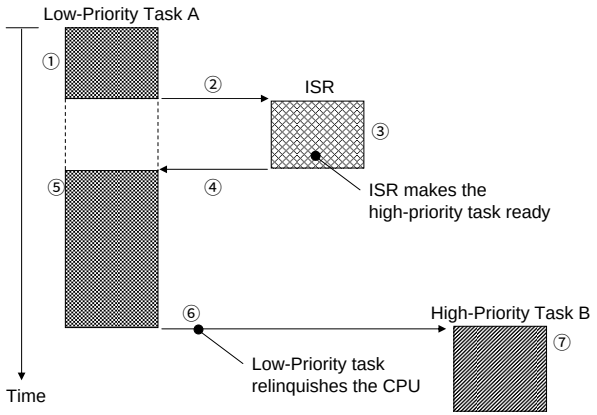
Context switch Diagram



Non-preemptive Kernel

- 비선점형 커널은 어떤 한 task가 수행하고 있을 때 kernel이 그 task의 수행을 강제로 중지시키고 다른 task를 수행시킬 수 있는 능력이 없음
- 다른 말로 cooperative multitasking이라고도 함
- Real-time system에서는 사용될 수 없는 구조
  - ✓ 이 구조에서는 priority가 낮은 task의 수행에 의해서 높은 priority의 task가 무한정 기다릴 수 있는 상황이 발생할 수 있기 때문에 사용하기 어렵다
    - Ex) Windows 3.1

Non-preemptive Kernel



Non-Preemptive Kernel 사례

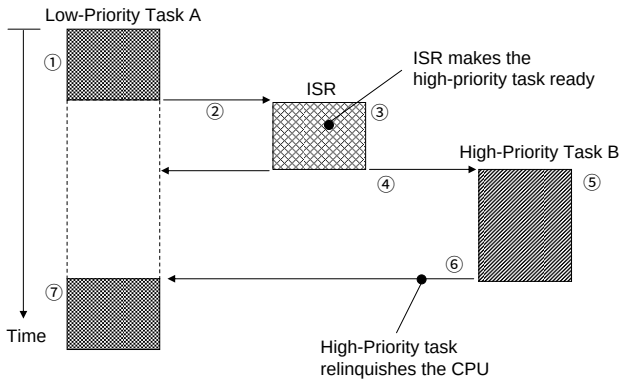
- 처리순서
  - ① Low priority taskA가 수행 중
  - ② interrupt 발생
  - ③ ISR에서 인터럽트 처리를 하고 Scheduler를 호출하면 현 task보다 높은 priority를 가진 task를 READY 상태로 만듦
  - ④ ISR의 수행이 종료되고 ISR 이전에 수행하던 Low priority taskA로 돌아감
  - ⑤ 인터럽트 발생 직전의 코드부터 다시 수행
  - ⑥ taskA가 system call을 호출하여 CPU 사용권 포기
  - ⑦ kernel에 의해서 taskB가 수행

  (앞의 예라면 serial 데이터에 대해서 처리 할 것임)

Preemptive Kernel

- 선점형 커널은 어떤 한 task가 수행하고 있는 도중에도 kernel이 그 task의 수행을 중지 시키고 다른 task (중지되는 task 보다 priority가 높은)를 수행시킬 수 있는 능력을 소유
- CPU를 사용할 준비가 된 가장 높은 priority의 task가 항상 CPU를 점유하여 수행될 수 있음
- Deterministic 한 특성을 가지고 있기 때문에 RTOS 에서는 이 구조를 사용
  - ✓ Ex) Windows 95/98/NT, UNIX

Preemptive Kernel



Preemptive Kernel 사례

① Low priority taskA가 수행 중

② 인터럽트 발생

③ ISR에서 인터럽트 처리를 하고 Scheduler를 호출하여 현 task 보다 높은 priority를 가진 task를 READY 상태로 만듦

④ ISR 종료와 동시에 ISR 이전에 수행하던 taskA가 아닌 high priority taskB가 CPU 제어권을 가짐

⑤ taskB 수행(앞에서 예를 든 serial 데이터에 대해서 처리 할 것임)

⑥ taskB가 kernel system call을 호출하여 CPU 제어권을 포기하고 kernel에 의해서 taskA가 선택됨

⑦ taskA의 인터럽트 발생 직전의 코드부터 다시 수행

Critical Section (Region)

- 다른 task에 의해서 중단되어선 안 되는 코드 블록
- 이 블록의 코드를 시작하게 되면, 코드 블록의 종료까지 Context Switch 에 의한 다른 task의 수행이 없어야 함
- Solution
  - ✓ Mutual Exclusion
  - ✓ Progress
  - ✓ Bounded Waiting
  - ✓ Semaphore

Mutual Exclusion

- 하나의 task가 공유자원을 사용하고 있는 동안 다른 task가 이 자원을 사용하지 못하도록 보장
- Mutual Exclusion을 보장하기 위한 방법
  - ✓ 공유자원을 사용하는 동안 인터럽트를 금지시킴

```
Disable interrupts;
    Access(read/write) the shared resource;
Enable interrupts;
```

  - 매우 간단
  - 인터럽트 금지 시간이 길어지면 문제 발생(인터럽트를 잃어버림)
  - 되도록 빠른 시간 안에 다시 인터럽트를 enable해야 함

Mutual Exclusion (2)

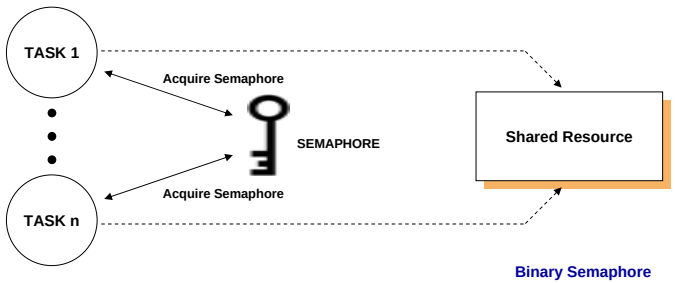
- ✓ 공유자원을 사용하는 동안 Scheduling을 금지

```
Disable scheduling;
    Access(read/write) the shared resource;
Enable scheduling;
```

  - Scheduling만을 금지 하였기 때문에 인터럽트가 발생될 수 있음
  - ISR에서 공유자원 접근 시 Mutual Exclusion을 보장할 수 없음
  - 따라서 task간에 공유되는 자원에 대해서만 사용됨
  - 이 방법을 많이 사용하게 될 경우에, 높은 priority task가 CPU를 점유할 수 있는 시점이 지연되기 때문에 deterministic의 특성이 파괴되는 문제가 발생
- ✓ Semaphore를 사용한다
  - 가장 좋은 방법
  - 공유 자원의 access time이 짧은 경우 위의 두 방법이 더 효과적

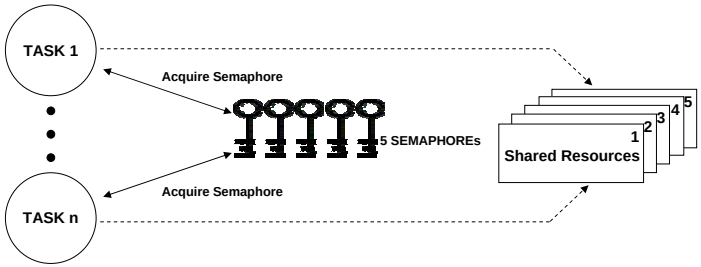
Semaphore

- Semaphore
  - ✓ 1960년대 중반 Edgser Dijkstra에 의해서 고안
  - ✓ 대부분의 RTOS에서 지원
  - ✓ 공유자원을 액세스하기 위해서는 key가 필요
  - ✓ Binary Semaphore



Semaphore (2)

- ✓ Counting Semaphore



- ✓ 반환된 Semaphore를 획득하기 위한 방법
  - priority based
  - FIFO based

Task Synchronization

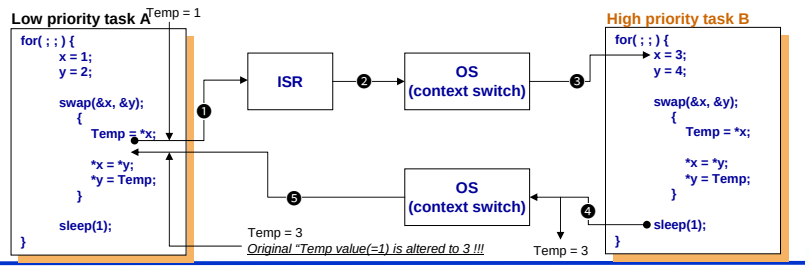
```
int    N = 0;

void taskA(void) /* task A */
{
       int    i;
       for (i = 1; i <= 2000;  i++)
              N++;
}

void taskB(void) /* taskB */
{
       int    i;
       for (i = 1; i <= 2000;  i++)
              printf("N is %d\n", N);
}
```

```
int    N = 0;

/*     semaphore X count = 0      */
/*     semaphore Y count = 1      */

void taskA(void) /* taskA */
 {
       int    i;
       for (i = 1; i <= 2000;  i++) {
              Take semaphoreX;  /* attempt to get semaphore X */
              N++;
              Give semaphoreY;   /* release semaphore Y  */
       }
}

void taskB(void) /* taskB */
{
       int    i;
       for (i = 1; i <= 2000;  i++) {
              Take semaphoreY; /* attempt to get semaphore Y*/
              printf("N is %d\n", N);
              Give semaphoreX; /* release semaphore X */
       }
}
```


Reentrancy

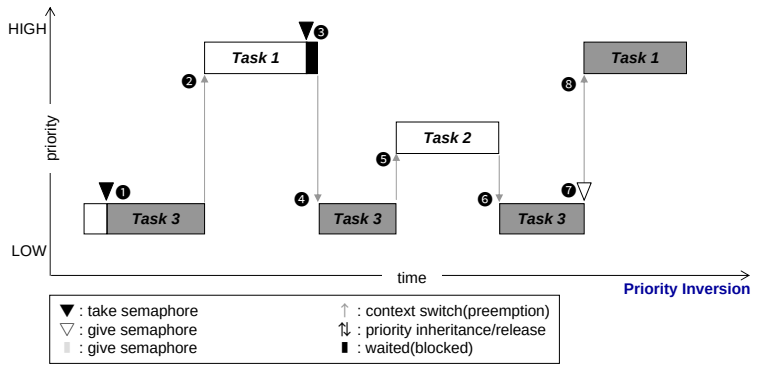
- ■ 코드의 재진입이 가능하다는 것을 의미
  - ✓ Non-reentrant function example

```
int Temp;
void swap(int *x, int *y)
{
       Temp    = *x;
       *x      = *y;
       *y      = Temp;
}
```



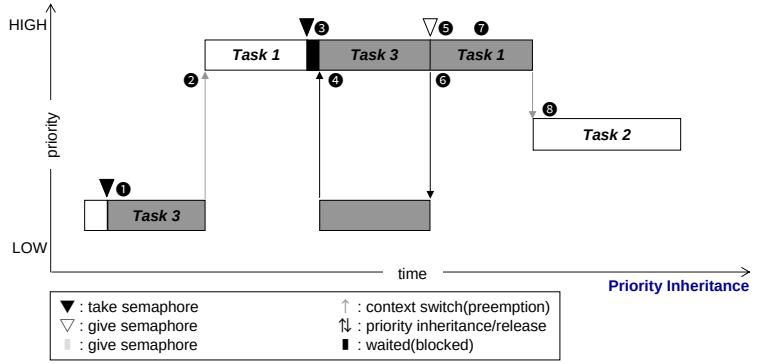
Priority Inversion & Priority Inheritance

- ■ Priority Inversion
  - ✓ 높은 priority의 task가 낮은 priority task의 수행이 끝날 때까지 기다리는 상황



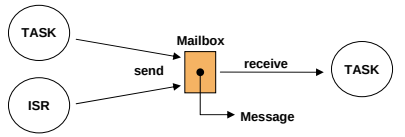
Priority Inversion & Priority Inheritance (2)

- Priority Inversion
  - ✓ 높은 priority의 task가 WAITING상태인 동안, 그 task를 기다리도록 만든 task의 priority를 높은 task priority레벨로 올리는 방법

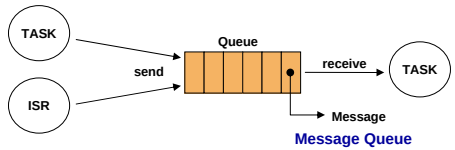


Task Communication (Inter)

- global variable
  - ✓ Linked list, Circular queue ..
  - ✓ Mutual Exclusion이 보장되어야 함
- message passing
  - ✓ Message Mailbox
    - 하나의 메시지



  - ✓ Message Queue
  - ✓ 복수 개의 메시지



Interrupts

- 하드웨어 mechanism
- CPU에게 asynchronous events을 알리는데 사용
- 인터럽트 종료 후 동작
  - ✓ Non-preemptive kernel
    - ISR이 종료 되면 ISR전에 수행하던 task를 다시 수행
  - ✓ Preemptive kernel
    - ISR의 종료 시점에서, Scheduling이 일어남
- 관련용어
  - ✓ Interrupt Latency :

    Maximum amount of time interrupts are disabled +

    Time to start executing the first instruction in the ISR

Interrupts (2)

- Interrupt Response :

  Interrupt Latency +

  Time to save the CPU's context +

  Execution time of the kernel ISR entry function(preemptive kernel only)

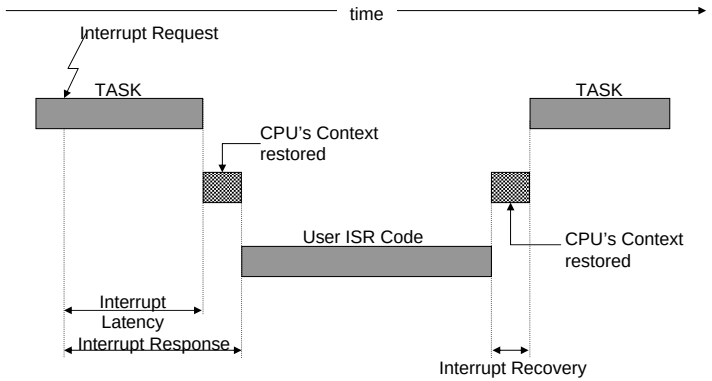
- Interrupt Recovery :

  Time to determine if a higher priority task is ready(preemptive kernel only) +

  Time to restore the CPU's context of the highest priority task +

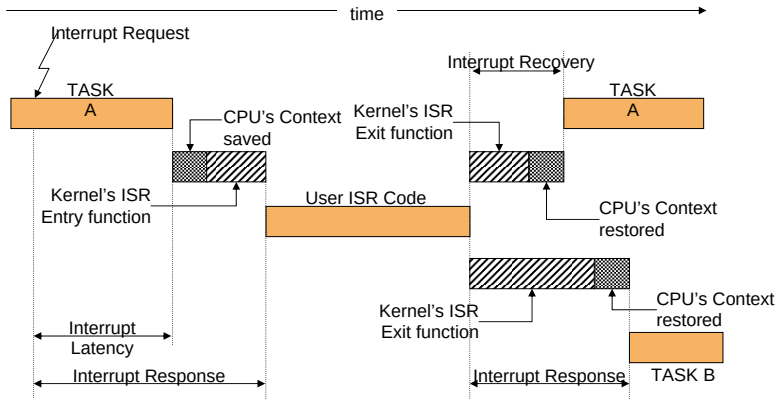
  Time to execute the return from interrupt instruction

Interrupts (3) non-preemptive kernel



Interrupt latency, response, and recovery (non-preemptive kernel)

Interrupts (4) preemptive kernel



Interrupt latency, response, and recovery (preemptive kernel)

RTOS 비교

- RTOS kernel의 interface를 중심으로 비교
- 비교 대상 운영체제
  - ✓ VxWorks
  - ✓ pSOS
  - ✓ Nucleus
  - ✓ VRTX
  - ✓ uC/OS II

RTOS 비교 - Task

- Task ID
  - ✓ Task ID – Task에 대한 Unique Key
    - VxWorks, pSOS, Nucleus
      - TCB의 pointer를 Task Id로 사용
      - OS가 TCB를 할당 받고, 그의 pointer값을 넘겨주는 형식
    - VRTX, uC/OS II
      - Virtual Task ID를 사용하고, TCB를 Table에서 찾아 사용
      - 0-255(VRTX), 0-63(uC/OS II)의 값을 사용
      - 사용자가 Task ID를 지정
- Task 수
  - ✓ 생성 가능한 Task의 수
    - VxWorks, pSOS, Nucleus, VRTX
      - Configuration에서 설정한 최대치 혹은 메모리가 가능한 만큼 생성이 가능하므로 Task 수의 제한이 없음
    - uC/OS II
      - 64개의 Task를 생성할 수 있으나, 상위 4개와 하위 4개의 Task를 OS가 사용하므로 실제 사용 가능한 Task 수는 56 개임

RTOS 비교 - Task

- Task 이름
- VxWorks, pSOS, Nucleus
- 이름을 지원하며 각각 10자, 4자, 8자 길이를 지원
- Task 이름은 사용자 입장에서 Task에 대한 unique ID 역할
- VRTX, uC/OS II
- Task의 이름을 따로 지원하지 않음
- Priority(우선순위)
- VxWorks, VRTX, Nucleus
- 0에서 255까지의 priority 지원(0이 가장 높음)
- pSOS
- 0에서 255까지의 priority 지원(0이 가장 낮음)
- 0과 240-255는 OS에서 사용
- uC/OS II
- Priority와 Task ID가 동일(0-63)

RTOS 비교 - Task

- Task 생성 단계
  - ✓ Nucleus, uC/OS II
    - 생성 후 즉시 Scheduling 시작
  - ✓ pSOS
    - 생성과 Scheduling 시작이 분리되어 있음
  - ✓ VxWorks, VRTX
    - 생성 후 즉시 Scheduling 시작되는 경우와 생성과 Scheduling 시작이 분리되는 2 가지 경우를 모두 지원
- Argument Passing(인수전달)
  - ✓ 생성되는 Task에게 data를 보내기위한 argument를 지원
  - ✓ VxWorks – 10개의 parameter 사용
  - ✓ VRTX – char *paddr과 unsigned long psize형태의 parameter block을 사용
  - ✓ Nucleus – argc와 argv형태 지원
  - ✓ pSOS – 4개의 integer argument 사용
  - ✓ uC/OS II – 3개의 parameter 사용

RTOS 비교 – Semaphore/Mutex

- Mutex 지원
  - ✓ Nucleus – 지원 안함
  - ✓ uC/OS II – 2.04 version부터 지원
  - ✓ pSOS – pSOS 3부터 지원
  - ✓ VxWorks – Binary Semaphore 형태로 지원
- Priority or FIFO
  - ✓ Semaphore는 pending되는 형태에 따라 Priority, FIFO가 있음
  - ✓ uC/OS II
    - Priority 만을 지원
  - ✓ VxWorks, pSOS, Nucleus, VRTX
    - Priority와 FIFO 모두 지원
    - Create interface시 선택할 수 있도록 함

RTOS 비교 – Semaphore/Mutex

- Name의 사용
  - ✓ pSOS, Nucleus – 이름 부여 가능
  - ✓ VxWorks, VRTX, uC/OS II – Name 지원 없이 Semaphore ID 사용
- Timout 중 No Wait 기능 형태
  - ✓ Semaphore Timout은 Timout, 무한대, No Timeout의 3가지
  - ✓ No Timeout – Semaphore를 얻을 수 없는 경우에 pend되지 않고 즉시 return 되는 형태를 말함
  - ✓ VxWorks, Nucleus
    - NOWAIT를 pending하는 interface의 timeout의 종류에 포함하는 방법을 사용 (FOREVER,NOWAIT,Timeout)
  - ✓ pSOS
    - WAIT, NOWAIT을 wait parameter에 사용하고, WAIT이라고 사용하는 경우에 0 혹은 Timeout값을 사용하도록 interface를 가져감
  - ✓ VRTX, uC/OS II
    - NOWAIT를 위해 accept()라는 interface를 추가적으로 사용

RTOS 비교 – Semaphore/Mutex

- 용어의 차이
  - ✓ Semaphore 경우
    - VxWorks – Take/Give 사용
    - pSOS – P/V 사용
    - VRTX, uC/OS II – Pend/Post 사용
    - Nucleus – Obtain/Release 사용
  - ✓ Mutex의 경우
    - VxWorks – Take/Give 사용
    - VRTX – Lock/Unlock 사용

RTOS 비교 – Queue

- Variable-length와 Fixed-length
  - ✓ Variable-length : queue의 message 크기가 다양
  - ✓ Fixed-length : queue의 message 크기가 일정
    - ➢ VxWorks – Variable-length 만을 지원
    - ➢ pSOS, Nucleus – Variable-length와 Fixed-length 모두 지원
    - ➢ VRTX, uC/OS II – Fixed-length 만을 지원
- Priority or FIFO
  - ✓ Queue에 pending되는 형태에 따라 나누어짐
  - ✓ uC/OS II
    - Priority 만을 지원
  - ✓ VxWorks, pSOS, Nucleus, VRTX
    - Priority와 FIFO 모두 지원
    - Create interface시 선택할 수 있도록 함

RTOS 비교 – Queue

- Name의 사용
  - ✓ pSOS, Nucleus – 이름 부여 가능
  - ✓ VxWorks, VRTX, uC/OS II – Name 지원 없이 Queue ID 사용
- Send/Post Timout 사용
  - ✓ Queue가 Full인 경우에 추가로 Send/Post하게 되는 경우에 error를 return하지 않고, 기다렸다 send/post하는 기능을 말함
  - ✓ VxWorks, Nucleus - 지원
  - ✓ pSOS, VRTX, uC/OS II – error를 return

RTOS 비교 – Queue

- Timout 중 No Wait 기능 형태
  - ✓ Queue Timout은 Timout, 무한대, No Timeout의 3가지 지원
  - ✓ No Timeout – Queue에서 message를 받을 수 없는 경우에 pend되지 않고 즉시 return되는 형태를 말함
  - ✓ VxWorks, Nucleus
    - NOWAIT를 pending하는 interface의 timeout의 종류에 포함하는 방법을 사용 (FOREVER,NOWAIT,Timeout)
  - ✓ pSOS
    - WAIT, NOWAIT을 wait parameter에 사용하고, WAIT이라고 사용하는 경우에 0 혹은 Timeout값을 사용하도록 interface를 가져감
  - ✓ VRTX, uC/OS II
    - NOWAIT를 위해 accept()라는 interface를 추가적으로 사용
- Broadcast 기능
  - ✓ Queue에서 pend되는 모든 Task에게 동일한 메시지를 보내는 기능
  - ✓ pSOS, VRTX, Nucleus– 기능 지원
  - ✓ VxWorks, uC/OS II – 직접 지원하지는 않음

RTOS 비교 – Queue

- 용어의 차이
  - ✓ Queue 경우
    - VxWorks, pSOS, Nucleus – send/receive 사용
    - VRTX, uC/OS II – post/pend 사용

RTOS (pSOS)

- 특징
  - ✓ Integrated Systems사에서 판매 했었으나 WindRiver에서 인수
  - ✓ 과거에 우리나라 여러 업체가 채택사용 (삼성전자 pSOS+ 개발에 참여 라이센스 보유)
  - ✓ 다른 RTOS보다 높은 기술력 보유
    - 컴파일러 전문업체와 디버깅 전문업체 따로 보유
  - ✓ 통합 개발환경 제공(pRISM+)
  - ✓ Kernel을 중심으로 여러 개의 Software component로 구성
    - 독립 모듈화
  - ✓ Kernel 사용료, application 사용료가 있으며, 개당 royalty
  - ✓ 그래픽 환경이 빈약함
    - 서버용 application에 적합

RTOS (VxWorks)

- 특징
  - ✓ WindRiver사에서 판매하는 제품으로 세계 시장 점유율이 가장 높음
  - ✓ 많은 종류의 마이크로 프로세서를 지원하며 대부분의 상용 Chip에 대한 Device Driver도 모두 지원
  - ✓ pSOS와 유사
  - ✓ 200개 정도의 독립 모듈 지원 -> 필요한 모듈 선택사용
  - ✓ 현재의 RTOS들이 지원하는 거의 모든 서비스 지원
  - ✓ 통합 개발환경 제공(Tornado:토네이도)
  - ✓ Kernel 사용료, application 사용료가 있으며, 개당 royalty
  - ✓ 그래픽 환경이 뛰어남

Multi Thread 모델 OS의 종류 (1)

- OSE
  - ✓ Enea OSE Systems에서 개발, 판매하는 RTOS로서 국내보다는 세계시장에서 훨씬 높은 인지도와 점유율을 가지고 있음
- VRTX
  - ✓ 몇 년 전만 해도 국내에서 가장 높은 시장 점유율을 가졌던 Mentor Graphics 사의 RTOS. 지금은 국내에서도 판매량이 줄고 있음
- Nucleus Plus
  - ✓ Accelerated Technology 사에서 개발, 판매하는 RTOS
  - ✓ 다른 RTOS들과는 달리 Full Source Code를 제공하며, 제품 당 지불하는 Royalty가 없음
  - ✓ 국내에서는 휴대폰 단말기와 PDA등 50여종의 제품에서 사용되고 있으며, 우리별 1호, 2호에도 탑재되어 있다고 함
- SuperTask
  - ✓ US Software 사에서 개발,판매하는 RTOS. Nucleus와 마찬가지로 Source Code를 Open하며, No Royalty

Multi Thread 모델 OS의 종류 (2)

- MicroC/OS (uC/OS)
  - ✓ 최근에 학교를 중심으로 많이 사용하면서 널리 알려진 RTOS
  - ✓ Jean J. Labrosse라는 사람이 개발하여 배포한 작은 크기의 RTOS이며, 책을 구입하면 부록에 Source Code가 포함되는 형태로 판매되며, Royalty 역시 없음
  - ✓ 꾸준한 Upgrade를 통하여 많은 종류의 프로세서를 지원
  - ✓ 현재는 Upgrade된 uC/OS-II 를 개발하여 배포하고 있으며, 이 책은 국내의 대형 서점에서도 구입 가능
- QNX
  - ✓ QNX Software Systems사에서 개발, 판매
  - ✓ 국내보다는 해외에서 많이 알려져 있고 시장 점유율도 높음
  - ✓ UNIX와 호환이 가능하며, 현재 비 상업용으로는 Real-Time Platform Package를 무료로 다운 받을 수 있음

Multi Process 모델 OS의 종류 (3)

- OS-9
  - ✓ Microware사에서 개발, 판매하는 RTOS
  - ✓ 국내 보다 세계시장에서 높은 인지도와 시장 점유율 가짐
- LynxOS
  - ✓ LinuxWorks사에서 개발, 판매하고 있는 Embedded Linux RTOS
  - ✓ UNIX와 호환이 가능하며 OS의 사이즈가 크고, 복잡하고 규모가 큰 Real-Time Application 개발에 적합
- RTLinux
  - ✓ Finite State Machine Labs 사에서 개발, 판매하는 Embedded Linux