論文 92-29B-4-2

한계수행시간에 기초한 실시간 자원관리 기법의 구현에 관한 연구

(A Study on the Implementation of a Real-time Resource Allocation Based Time-Constraints)

李 楨 培* 朴 容 震**

(Jeong Bae Lee and Yong Jin Park)

要 約

실시간 프로세스를 한계수행시간내에 수행을 완료하기 위해 실시간 운영체제에서는 여러 기법들을 사용하고 있다. 본 논문에서는 실시간 프로세스에게 지원되어야 하는 자원이 다른 프로세스에 의해 이미 점유될 때, 실시간 프로세스가 한계수행시간내에 수행 가능성을 높이기 위한 실시간 자원 관리기법을 제시하였다. 본 논문에서는 이러한 기법을 UNIX 운영체제에 구현하고 시험하였다.

Abstract

Many real-time supporting features are used to execute real-time process in a time-constraints given. In this paper, The real-tiem resources management mechanism based time constraints which order resources supporting real-time processes blocked by the processes was proposed. This mechanism was all implemented and tested on the UNIX operating system.

I. 서 론

과거의 UNIX 운영체제는 실시간 처리기능이 없다. 예를 들면, UNIX의 라운드로빈(round robin) 방식은 일반 프로세스에게 공정한 우선순위를 제공하면서 반복적으로 해당 프로세스를 선택하여 CPU의 서비스를 받도록 하기때문에 실시간 프로세스를 지원하지 못한다.[3,12]

일반적으로, 실시간 시스템은 강 실시간 시스템(hard real-time systems)과 약 실시간 시스템(soft real-time systems)으로 구분된다. 강 실시간 시스템은 요구되는 응답이 정확해야 하며 동시에 한정된 시간 간격내에서 응답이 있어야 한다. 이러한 시스템은 비행기 통제 시스템, 로보틱스, 핵 발전제어, 생산 제어공정 등 응답이 한계수행시간(time constraints)내에 반드시 제공되어야 하는 응용분야에서 많이 사용된다. 만약 응답이 한계수행시간내에 제공되지 않을 경우에는 지체없이 비상 대책 루틴을 수행하여 피해가 없도록 조치를 취해야 한다. 주로 약 실시간 시스템은 이러한 응답이 한계수행시간 내에 수행이 되지 않더라도 피해가 적은 응용분야에 사용된다. 일반적으로 실시간(real time)이란 그 시스템이 적용되는 응용 분야에 따라 다르며 실시간 운영체제의 구조 역시 적용분야의 환경에 따라 다양하다.[10,13]

오늘날의 UNIX 운영체제에서는 실시간 처리 기능

*正會員, 釜山外國語大學校 컴퓨터工學科
(Dept. of Com. Eng., Busan Foreign Studies Univ.)
**正會員, 漢陽大學校 電子工學科
(Dept. of Elec. Eng., Hanyang Univ.)

接受日字 : 1991年 11月 4日

을 부가시킨 시스템을 구성하여, 주로 약 실시간 시스템 관련 응용 분야에 사용할 수 있도록 하는 연구가 많이 추진되었다. 예를 들면, Masscomp사는 우선순위에 기초한 스케쥴링(priority based scheduling), 메모리 로킹(memory locking)기법, 인접 화일 시스템(contiguous file system) 기법을 기존 운영체제에 구현시킨 RTU(real-time UNIX)를 개발하였다. Action Instruments사는 IBM XT와 AT에서 수행이 가능하고 UNIX와 호환성이 있는 실시간, 다중 타스킹(multitasking) 운영체제인 IC-DOS를 개발 하였다. 또한 HP사는 우선순위에 기초한 스케쥴링, 디스크 영역의 예약 할당(pre-allocation) 등의 기법을 구현한 HP-UX가 많은 응용 분야에 사용되고 있다. 그 외에 AT & T사의 DMERT, MERT와 Diab System사의 D-NIX 등 다수의 실시간 UNIX 운영체제가 개발되었다. 이와 같이 다양한 실시간 기능들을 선택적으로 UNIX 운영체제에 추가하여 실현시킬 수 있다.[1,2,9,14]

본 논문에서는 이러한 UNIX 환경에서 실시간 프로세스에게 요구되는 자원들이 다른 프로세스에 의해 점유됨으로 인하여 실시간 프로세스가 오랫동안 block되는 단점을 보완하기 위하여 한계수행 시간에 기초한 자원 점유의 공고, 한계수행시간의 전파, 한계수행시간에 기초한 자원 방출 알고리듬을 제시한다. 이 기법은 주어진 자원 테이블의 한계수행시간을 참조하여 우선순위를 정하며, 프로세스의 수행시간을 검계하여 수행시간이 지난 실시간 프로세스를 종료시키는 기법을 제시한다. 또한 실시간 프로세스의 수행시간을 검사하는 기능을 제공하여 한계수행시간을 만족하지 못할 경우에는 다른 조치를 취할 수 있도록 하였다.

Ⅱ. 한계수행시간에 기초한 실시간 프로세스를 위한 자원 할당

일반적으로 긴급한 한계수행시간을 요구하는 높은 우선순위를 갖는 실시간 프로세스가 보다 낮은 우선순위의 프로세스를 호출하였을 경우, 그 프로세스의 수행이 끝나서 복귀하기 전까지는 낮은 우선순위를 가져야 하는 우선순위 반전(priority inversion) 현상이 발생할 수 있다. 이는 Fig. 1에서 보는바와 같이 "10"이라는 높은 우선순위의 실시간 프로세스가 "50"이라는 낮은 우선순위의 프로세스를 호출하였을 경우, 호출된 프로세스가 낮은 우선순위의 프로세스일 경우에 높은 우선순위의 실시간 프로세스가 호출한 프로세스의 수행이 끝날때까지 기다려야 하는 문제

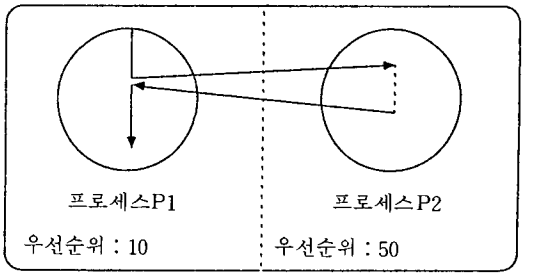


그림 1. 우선순위 반전 현상
Fig. 1. The state of priority inversion.

가 발생될 수 있다는 것을 의미한다.[6]

이러한 문제를 해결하기 위해 우선순위 상속(priority inheritance)이라는 기법에 의해 "10"의 우선순위를 갖는 실시간 프로세스가 호출한 프로세스가 "50"이라는 낮은 우선순위를 갖는 프로세스를 호출할 경우 "10"의 우선순위를 호출된 프로세스에게 그대로 물려줄 수 있도록 하여야 한다. 마찬가지로 실시간 프로세스가 사용하고자 하는 다른 자원이 다른 프로세스에 의해 사용되고 있을 경우에도 실시간 프로세스는 그 자원이 방출되기를 기다리며 block이 될 수 있다. 이 경우에는 실시간 프로세스가 정의된 한계수행시간을 만족시켜줄 수가 없게 된다. 이러한 면을 보완하기 위하여 본 논문에서는 실시간 프로세스가 요구하는 자원을 빠른 시간내에 가용한 상태로 전환하기 위한 한계수행시간에 기초한 실시간 자원 관리기법을 제시한다.

이 기법은 프로세스에게 주어진 한계수행시간에 따라 프로세스의 우선순위를 정하여 낮은 우선순위의 프로세스에게 이미 할당된 자원을 높은 우선순위의 실시간 프로세스가 그 자원을 요구할 경우, 그 자원이 가능한 한 빠른 시간내에 방출되게 하여 실시간 프로세스가 block되는 시간을 감소시키는 방법을 사용하고 있다.

1. 한계수행시간의 관리

본 논문에서는 프로세스 테이블에 한계수행시간 관련 정보란을 추가하여 실시간 프로세스가 만들어질 때 프로세스 테이블에 실시간 관련 정보를 기록하는 방법을 사용하였다. 즉, 프로세스 테이블에는 한계수행시간 란과 이에 대응되는 실시간 우선순위 란을 만들어 프로세스 스케쥴링시에 참조되도록 하였다. 한계수행시간은 시스템에 따라 의존적이어서 최대로 지원할 수 있는 한계수행시간이 제한적으로 제공될 뿐만 아니라, 적용되는 응용분야에 따라 제공

되어야 하는 한계수행시간이 다르기 때문에 한계수행시간과 관련된 변수의 정의는 수퍼 유저가 조정해 주어야 한다.

UNIX에서 프로세스 스케쥴링은 우선순위를 기반으로 하고 있다. 본 논문에서는 이러한 특성을 실시간 프로세스에 적용을 하기 위하여 Fig. 2에서 보는 바와 같이 프로세스의 수행은 한계수행시간에 의해 결정된 우선순위에 의해 실시간 프로세스의 우선순위가 정해짐으로써 결정되도록 하고 한계수행시간의 긴급성에 따라 우선순위가 결정되도록 하였다. 한계수행시간의 종류는 실시간 환경에 따라 다를 수 있으므로 헤더화일(header file)에 정의함으로써 수정이 가능하다.

그러므로, 실시간 프로세스 스케쥴러는 run-큐(queue)에 있는 프로세스들 중에서 항상 가장 높은 우선순위를 가진 프로세스를 다음에 switch-in될 프로세스로 선정한다. 물론 같은 우선순위를 프로세스는 라운드-로빈 방식에 의한다.[5]

	실시간 우선순위	한계수행시간
[0]	100	100ms
[1]	101	98ms
[2]	103	96ms
⋮		
[49]	150	10ms

그림 2. 한계수행시간에 따른 실시간 우선순위 할당 예

Fig. 2. An example of the real-time resource allocation based time constraints.

시스템에서는 해당 프로세스가 실시간 프로세스의 여부를 이 프로세스 테이블에 기록된 한계수행시간을 참조하여 결정한다. 프로세스 테이블에는 한계수행시간에 따른 실시간 우선순위로 변환 기능을 통하여 결정된 실시간 우선순위를 기록함으로써 프로세스 스케쥴링시에 참조하여 우선적인 처리가 가능하도록 한다. 하나의 프로세스가 실시간 프로세스로 수행되게 하는 방법은 부팅(botting)시에 결정되어 수행되게 하는 방법이 있을 수 있고 시스템-호출에 의한 것일 수 있는데 여기서는 일반 사용자 프로그래머가 실시간 프로세스를 지정할 수 있도록 후자의 방법을 채택하였다. 시스템-호출에 의한 운영체제의 접근은 Fig. 3에서 보는 바와 같다. 이러한 특정

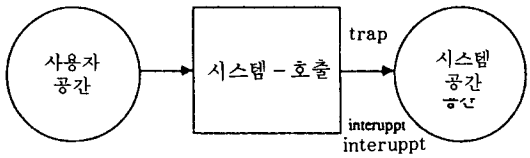


그림 3. 시스템-호출에 의한 운영체제의 접근

Fig. 3. The access of the operating system by the system call.

프로세스의 한계수행시간을 제어하는 시스템-호출은 수퍼 유저(super user)에게만 허용되게 하며 이 시스템-호출의 인수(argument)로서 프로세스-고유번호(process-id)와 한계수행시간이 필요하다. 한계수행시간은 사용되는 시스템에 따라 다르며 한계수행시간의 종류는 변수들을 수퍼 사용자가 정의하여 사용할 수 있다. 단, 프로세스-고유번호가 정의되어 있지 않은 경우에는 현 프로세스를 의미한다. 이 시스템-호출의 알고리듬을 C-pseudo 코드에 의하여 나타내면 Fig. 4에서 보는바와 같다.

```
/*
 * 한계수행시간 (time-constraints) 설정 시스템-호출
 *
 */
rtpio(process_id, ts)
{
  if (not super-user)
    return;
  if (process_id= =0) /* Current proc. itself */
    set the ts as defined; /* ts;time-constraints */
  else{
    find the process table defined;
    set ts to the process table;
    set real-time priority by converting ts to priority;
    if (ts of defined proc>ts of running proc)
      runrun++;/* set rescheduling flag */
  }
}
```

그림 4. 한계수행시간 설정 시스템-호출

Fig. 4. The system call to allocate a time constraints.

이러한 시스템-호출에 의하여 한계수행시간을 부여받은 실시간 프로세스는 해당 우선순위를 부여받아 실시간 처리가 되겠으나, 만약 이 프로세스가 이용하여야 하는 자원이 다른 프로세스에 의해 block되어 있다면, 그 자원이 가능한 한 빠른 시간내에 방출되어 실시간 프로세스에게 제공되도록 한다. 이러

한 문제를 해결하기 위한 기법이 본 논문에서 제시되는 실시간 자원 관리 기법이다. 이 기법에서는 실시간 프로세스의 수행이 한계수행시간을 만족하지 못할 경우에는 특별한 조치를 취할 수 있도록 한계수행시간 추적 기능을 제공하고 있다.

Ⅲ. 실시간 자원관리 기법

자원이 가용하지 않음으로 인해 실시간 프로세스가 block 되는 경우에 그 자원이 빠른 시간내에 방출되게 함으로써 block되어 있는 실시간 프로세스가 wake-up되어 수행이 될 수 있도록 지원하는 실시간 자원관리 기법을 본 논문에서 제시한다.

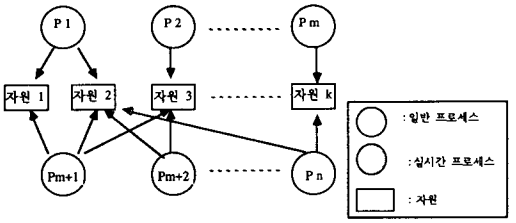


그림 5. 실시간 자원 관리 기법
Fig. 5. The mechanism for managing the real-time resources.

Fig. 5에서 보는바와 같이 임의의 프로세스 P1, P2, …, Pm이 여러 자원을 점유하여 사용하고 있을 경우, 실시간 프로세스 Pm+1은 P1이 점유하고 있는 자원1을 요구할 수 있다. 이러한 경우에 Pm+1은 P1이 자원1을 방출할때까지 기다려야 한다. 실시간 프로세스 Pm+1을 위해서 자원1을 점유하고 있는 프로세스 P1이 자원1을 빠른 시간내에 방출하는 방법은 아래의 두 방법이 있다.

(1) 프로세스 P1을 선점하여 프로세스 Pm+1에게 강제로 자원을 방출되게 하는 방법

(2) 프로세스 P1에게 프로세스 Pm+1의 한계수행시간을 상속하여 프로세스 P1을 실시간 프로세스화 함으로써 프로세스 P1의 수행이 가능한 한 빨리 종료가 되도록 하여 자원을 방출하도록 하는 기법

(1)의 방법은 주로 강 실시간 시스템에 많이 사용되는 기법으로 실시간 프로세스에게 절대적인 우선권을 제공한다. 그러나, 프로세스 P1이 대부분 수행을 완료하였을 경우에는 비효율적인 수행이 되기 때문에 단점이 된다. (2)의 방법은 실시간 프로세스 P1의 수행이 거의 종료되었을 경우에는 효율적인 방법이 되나 한계수행시간을 만족시키기 위해서는 비효율적인 방법이므로 주로 약 실시간 시스템에 많이 사용된다. 본 논문은 2)의 방법을 이용하여 구현하였다.

이러한 환경에서는 실시간 프로세스 Pm+1이 block될 경우, 자원1이 프로세스 P1에 의해 점유되어 있다는 사실을 알 수 있어야 한다. 이러한 문제를 해결하기 위한 기법은 어떤 프로세스가 하나의 자원을 점유할 때마다 그 정보를 자원 테이블에 공고하여 그 정보를 커널에서 참고할 수 있어야 한다. 커널은 실시간 프로세스 Pm+1을 block되게 한 프로세스 P1를 확인할 수 있으며, 프로세스 P1의 한계수행시간을 Pm+1만큼 일시적으로 높여주고, 더욱 더 긴급한 실시간 프로세스 Pm+2가 임의의 자원3을 요구할 경우에는 프로세스 P1에게 실시간 프로세스 Pm+2의 한계수행시간을 상속시킨다.

여기서 원하는 스케쥴링은 이루어지지만 또 하나의 새로운 문제가 발생된다. 즉, 프로세스 P1이 자원3을 방출한 후에는 프로세스 Pm+1의 한계수행시간을 보유하여야 하고 자원2를 방출한 후에는 원래 자신의 한계수행시간을 보유하여야 한다. 이러한 요구를 실시간 자원 관리 기법에서는 모두 지원하여야 한다. 또한, 자원2가 방출되더라도 프로세스 Pm+1, Pm+2, …Pn이 자원 2를 점유하기 위한 경쟁상태에 들어 간다. 이러한 경우에는 가장 한계수행 시간이 긴급한 프로세스가 우선적으로 자원을 점유한다.

위에서 설명한 한계수행시간에 기초한 자원관리 기법을 자세히 설명하면, 다음과 같이 한계수행 시간에 기초한 자원 점유의 공고 알고리듬, 한계수행시간 전파 알고리듬, 한계수행시간에 기초한 자원 방출 알고리듬으로 나눌 수 있다.

1) 한계수행시간에 기초한 자원점유의 공고

한계수행시간에 기반을 둔 자원점유 공고 알고리듬을 C-Pseudo 코드로 작성하면 다음과 같다.

이 알고리듬은 프로세스가 하나의 자원을 점유하게될 때, 그 자원의 점유됨을 공고하기 위한 것이다. 자원 테이블 목록(resource table list)으로 부터 자유-노드(free node) 하나를 할당하여 자원-고유번호(resource-id), 소유자 프로세스(owner process), 소유자 프로세스의 한계수행시간 등으로 정보를 채운 뒤, 해쉬 큐(hash queue)에 등록한다. 자원점유의 공고 알고리듬은 Fig. 6에서 보는 바와 같다. 여기에서 Fig. 7에서 보는 바와 같이 한계수행시간을 자원 테이블에 등록시킴으로써 한계수행시간 추적을 가능하도록 한다.

```
broad_rsc(resource_id)
{
  if (resource_id= =0)
      error("addrscq:resource_id is zero");
  /*
   * 자유-노드의 할당
   */
  LOOP
    access the designated resource-table;
    if(resouce_table is empty)
      break;
    increase the pointer of resource-table;
  END_LOOP
  enter the resource_owner to the resource_table;
  /* resource_owner:resource_id에 해당하는 자원을
     사용하는 프로세스 */
  register ts to the resource_table;
  enter the resource_id to the resource_table;
  enter the resource_table to the resource queue
  by using hash_key;
}
```

그림 6. 한계수행시간에 기초한 자원점유의 공고 알고리듬

Fig. 6. The algorithm to broadcast the held resources based time constraints.

2) 한계수행시간의 전파

이 알고리듬은 실시간 프로세서가 가용한 자원을 기다리기 위해 block될 경우, 이미 그 자원을 기다리고 있던 모든 프로세스들에게 자신의 한계수행시간을 전파하기 위한 것이다. 우선 실시간 자원 관리 기법을 위해서 자원 테이블의 구조에 추가된 항목은 Fig. 7에서 보는 바와 같다.

한계수행시간 추적 버퍼 주소
한계수행시간
자원의 고유번호
자원의 소유자 프로세스
실시간 프로세스의 갯수
다음 자원 테이블의 pointer

그림 7. 자원 테이블의 구조

Fig. 7. The structure of the resource table.

이 알고리듬에서는 원하는 자원에 관한 정보를 자원 테이블 목록에서 참조하여 현 실시간 프로세스의 한계수행시간이 그 자원으로 인해 block되어 있는 다른 실시간 프로세스에 의해 보다 긴급한 한계수행시간이 자원 테이블에 등록되어 있으면 아무런 조치를 취하지 않는다. 그러나, 위의 경우가 아닌 경우에는 자원 테이블의 한계수행시간을 현 실시간 프로세스의 한계수행시간으로 천이시킨다. 이상의 전파 알고리듬을 C-pseudo 코드에 의해서 나타내면 Fig. 8에서 보는 바와 같다.

```
/* current_proc:전파를 위한 실시간 프로세스 */
/* resource_id:current_proc가 필요로 하는 자원의 주소*/
/* owner process:자원의 소유자 프로세스 */
xl_propagate(current_proc)
{
  while(current_proc= = 'BLOCK'){
    hask_key=HASH(resource_id);
    find the resource needed from resource queue by
    using hash_key;
    if(current_proc_ts<=resource_ts)
      return;
    else
    increment the resource_ts to current_proc_ts;
  }
}
```

그림 8. 한계수행시간 전파 알고리듬

Fig. 8. The algorithm to propagate the time constraints.

3) 한계수행시간에 기초한 자원 방출

이 알고리듬은 프로세스가 자신이 점유하고 있던 자원을 방출(release)할 경우, 그 자원에 관한 정보를 자원 테이블 큐(resource table-queue)로 부터 제거하고 다음의 낮은 한계수행시간으로 천이가 된다. 제거된 자원 테이블은 empty, 즉 '0'으로 리세트(reset)된다. 이 과정에서 여러개의 자원을 block하고 있던 프로세스가 하나의 자원만을 방출할 경우에, 그 자원 테이블의 한계수행시간을 참조하여 해당 실시간 프로세스와 연관된 수를 '1' 감소시킨 결과가 '0' 이상이면, 그 한계수행시간을 계속 유지한다. 그렇지 않을 경우에는 다음 '0'이 아닌 한계수행시간으로 천이한다. 한계수행시간에 기초한 자원 방출 알고리듬은 Fig. 9에서 보는바와 같다.

4. 한계수행시간 추적 기법

한계수행시간 추적을 위해서 기본적으로 UNIX 에서의 프로세스 상태들의 종류와 천이 과정을 분석하

```
/* rs_owner:자원의 소유자 */
del_rscq(resource_id)
{
  find resource to be deleted by using link in the
  resource_que;
  if(resource_id->rs_ower ! =current_proc)
      error("resource:not owner");
  clear the resource table to mak  "empty"; mark
  ts_count=ts_count-1;
  if(ts_count<=0)
    change resource_ts to next resource_ts;
  else
    return;
}
```

그림 9. 한계수행시간에 기초한 자원 방출 알고리듬
Fig. 9. The algorithm to release resources based time constraints.

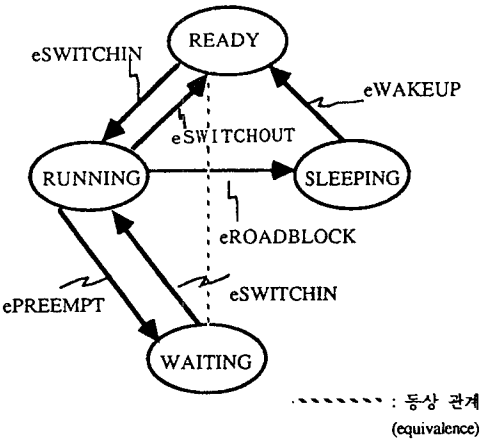


그림10. 프로세스의 상태 천이도
Fig. 10. The diagram of the state transition of the process.

여야 한다. 프로세스는 4종류의 프로세스 상태(RUNNING, SLEEPING, READY, WAITING)를 가지게 되며 각각 프로세스 상태에서 소요된 시간을 추적하여 기록, 관리할 수 있어야 한다. 사용자에게는 이러한 기능을 사용할 수 있도록 시스템-호출을 제공한다. 4가지 프로세스 상태에서 각각 소요된 시간은 밀리초(milisecond) 단위로 측정할 수 있도록 한다. 여기에서 측정할 수 있는 최소 단위는 시스템에서 제공하는 1clock-tick이다. 추적이 될 4가지 프로세스 상태의 천이를 결정하는 사건 인자는 Fig. 10에서 보는바와 같이 switch-in, wakeup, preempt, block이다. 따라서, 프로세스 상태 천이가 일어나는 이 4종류의 상태에서 시간 추적이 가능하다. 그리하여 정의된 시간 이내에 자원 관리가 되지 않는다면 사용자는 적절한 조치를 취할 수 있도록 시스템-호출을 지원한다. 이는 사용자가 한계수행시간 추적 시스템-호출을 통하여 리턴되는 값을 보고서 결정할 수 있다.

가) 한계수행시간 추적을 위한 사용자 인터페이스

한계수행시간 추적은 UNIX 커널 내에서 이루어지기 때문에 사용자에게 이를 접근할 수 있도록 사용자 인터페이스를 제공하여야 한다. 이를 위해서 두가지 종류의 시스템 호출을 제공하는데 이를 나열하면 다음과 같다.

1) 한계수행시간 추적을 구동하는 시스템-호출: begin_trace()

한계수행시간 추적을 구동하기 위한 시스템-호출로서 UNIX 커널 내의 한계수행시간 추적을 위한 함수를 수행하도록 한다. 한계수행시간의 추적은 위에서 설명한 각 프로세스 상태에서 체류한 시간을 유지 관리한다.

2) 한계수행시간 추적결과를 얻는 시스템-호출: get_trace()

한계수행시간 추적 결과치를 얻는 시스템-호출로서 UNIX 커널 내의 한계수행시간 추적 함수에서 유지 관리한 추적 결과 시간을 구한다. 한계수행시간 추적 결과치는 각 프로세스 상태에서 체류한 시간을 모두 합한 값이다. UXIN 커널에서는 이 시스템-호출에 의해 추적 결과치를 사용자에게 돌려 준다.

사용자는 begin_trace() 시스템 호출을 실시간 응용의 시작점에서 수행시키고 원하는 루틴에서 get_trace() 시스템-호출로 한계수행시간 추적 결과치를 얻을 수 있다. 사용자는 추적 결과치를 분석하여 한계수행시간을 경과했을 경우에 긴급처리를 위한 함수를 호출하여 응급처리를 할 수 있다. UNIX 커널에서 한계수행시간 추적을 위해 요구되는 데이타 구조들을 도해하면 Fig. 11에서 보는바와 같다.

사용자가 UNIX 커널내의 한계수행시간 추적 함수를 접근하는 과정은 Fig. 12에서 보는 바와 같이 프로세스의 각 상태에서의 소요시간을 기록 유지하고 있다가 한계수행시간 추적 결과치를 요구하는 시스템-호출이 발생되었을 때, 프로세스가 수행된 경과 시간을 리턴함으로써 가능하다. 이때, 리턴되는 추적 결과 정수가 "-1"이면 예외취급함수(exception handler)를 호출하여 예외적인 사건에 대한 긴급처리를 할 수 있다. 사용자 프로그램은 rtpio( )시스템 호출을 이용한 실시간 프로세스이며 이 프로그램의 시작 부분에

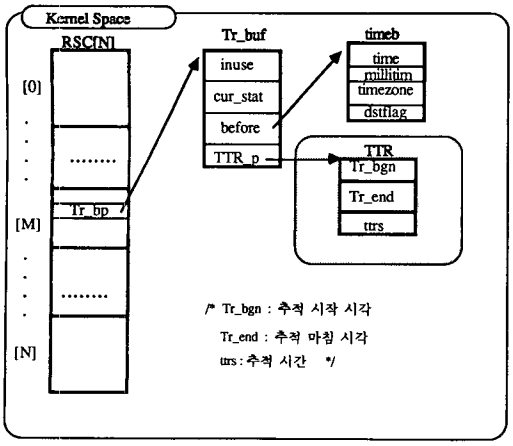


그림11. UNIX 커널에서 한계수행시간 추적을 위한 데이타 구조

Fig. 11. The data structure to trace time constraints on the unix kernel.

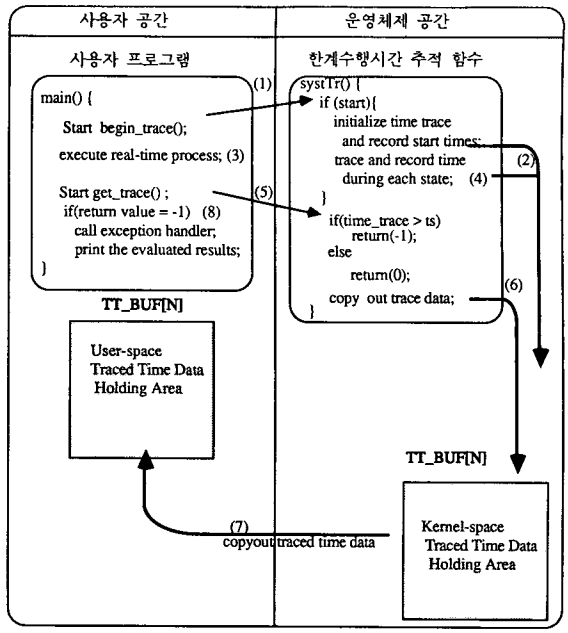


그림12. 한계수행시간 추적 과정

Fig. 12. The procedure of the time constraints trace.

한계수행시간 추적 시작 시스템-호출인 begin-trace ( )를 사용하고 마지막 부분에 한계수행시간 추적 결과치를 얻는 시스템-호출인 get-trace( )를 이용한다. 이러한 시스템-호출은 이 실시간 프로세스가 Fig. 10에서 보는 바와 같이 프로세스의 각 상태에서 머문 시간을 추출할 수 있도록 환경을 제공한다. 모든 프로세스는 이러한 4가지 상태에서 머물기때문에 각 상태에서 머문 총 체류시간이 시스템에서 소요된 시간이다. 이 시간을 한계수행시간과 비교하여 이를 초과한 경우에는 필요한 조치를 취할 수 있다. 이러한 조치는 약 실시간 시스템에서 효과적으로 이용될 수 있으나, 강 실시간 시스템을 위해서는 보다 개선된 기법이 요구된다.

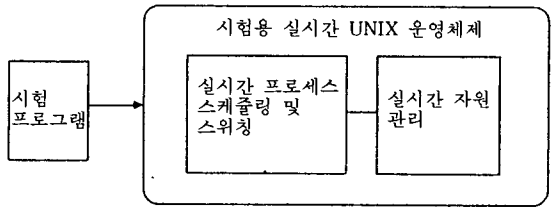


그림13. UNIX에서 실시간 처리 시험 환경

Fig. 13. The test environment of the real-time processing on the unix.

5. 시험 및 고찰

운영체제에서 실시간 처리 기법을 시험 및 성능 평가를 위한 일반적인 도구는 공개되어 있지 않으므로 본 시험에서는 실시간 자원 관리 기법을 위한 시험 프로그램을 작성하여 시험하였다. 이러한 시험 프로그램 결과를 평가하기 위해서 Fig. 13에서 보는 바와 같이 실시간 처리 기능이 없는 일반 UNIX 운영체제에 실시간 프로세스 스케줄링 기능과 실시간 입출력 요구를 우선적으로 처리할 수 있는 실시간 디스크 스케줄링 기법을 추가시켜 시험 환경을 구축하였다. 실시간 프로세서는 Fig. 4에서 본 바와 같이 한계수행시간 설정 시스템-호출(rtpio( ))에 의해 만들어 진다.

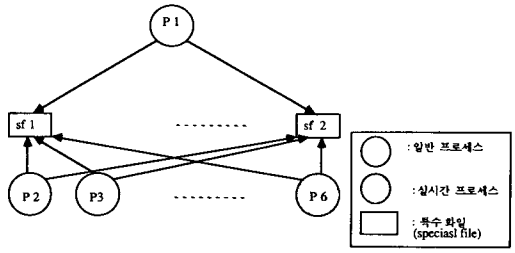


그림14. 시험 프로그램 구조

Fig. 14. The structure of the test program.

시험 프로그램의 구성은 Fig. 14에서 보는 바와 같이 일반 프로세서가 자원으로서 특수 화일인 디스크 장치를 선정하여 특정 화일을 한 디스크 장치에서 읽어 들여 다른 디스크 장치에 기록하는 동안에 다른 실시간 프로세스와 일반 프로세스들을 생성시켜 일반 프로세스가 이미 점유하고 있는 특수 화일인 디스크 장치들을 요구하는 시험 프로그램을 작성하여 시험하였다. 즉, 프로세스 P1은 각각 다른 화일 시스템인 특수 화일 sf1, 과 sf2를 선정하여 sf1에서 데이타를 읽어 들여 sf2에 기록을 하는 상태를 기본적으로 구축한다. 이 상태에서 sf1과 sf2를 요구하는 일반 프로세스(P1-P5)가 이미 block되어 있는 상태에서 실시간 프로세스 P6를 생성시켜 sf1과 sf2를 요구하는 시험 환경을 구축하였다.

이 시험 프로그램의 시험을 위하여 구축한 실시간 운영체제에 입력시켜 수행된 결과치를 도표로 나타내면 Fig. 15, Fig. 16에서 보는 바와 같다. Fig. 15에서는 실시간 프로세스가 이미 block된 일반 프로세스가 5개 존재하는 상태에서 발생하여 처리된 시험 결과이다. Fig. 16은 일반 프로세스가 특수화일을 요구하여 이미 block된 프로세스가 5개 존재하는 상태에서 측정한 시험 결과이다.

시험 프로그램에서 사용되는 프로세스의 수행시간 측정은 Fig. 12에서 제시한 한계수행시간 관련 시스템-호출을 이용하였다. 즉,운영체제 공간에서 기록된 프로세스의 각 상태에서 측정된 시간을 get-trace( ) 시스템-호출을 이용하여 사용자 공간의 버퍼에 기록된 값을 프로세스의 수행시간으로 추출하였다. 시험 프로그램의 종류는 두가지로서 하나는 rtpio( ) 시스템-호출을 이용한 실시간 프로세스이고, 다른 하나는 이 시스템-호출을 이용하지 않은 일반 프로세스이다. 즉 실시간 프로세스가 block된 자원을 요구하는 경우에는 운영체제내에 구현된 실시간 자원 관리 기법의 지원을 받게 된다.

시험 결과를 고찰하여 보면, 위의 도표에서 보는 바와 같이 실시간 자원 관리 기법을 추가시킨 결과로 Fig. 15에서와 같이 이미 일반 프로세스에 의해 점유된 자원을 차지하기 위해서 다른 일반 프로세스들과의 경쟁에서 절대적인 우위를 차지함을 알 수 있다. Fig. 16을 살펴보면, 일반 프로세스는 자신이 요구하는 자원들이 이미 다른 프로세스에 의해서 점유되어 있는 자원이 많으면 많을수록 block되어 있는 시간이 길어짐을 볼 수 있으나, 실시간 프로세스의 block 되는 시간은 거의 일정함을 볼 수 있었으며, 한계수행시간을 초과한 실시간 프로세스에 대해서 예외처리 함수를 호출함으로써 예외적인 상황에 대한 긴급 처리가 가능함을 볼 수 있다.

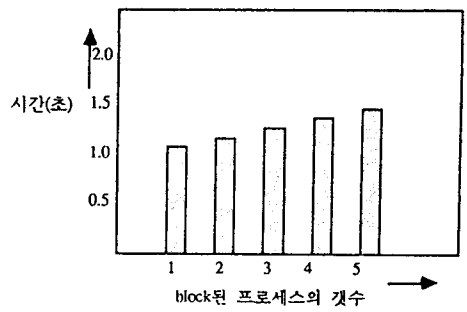


그림15. 실시간 프로세스의 응답시간
Fig. 15. The response time of the real-time process.

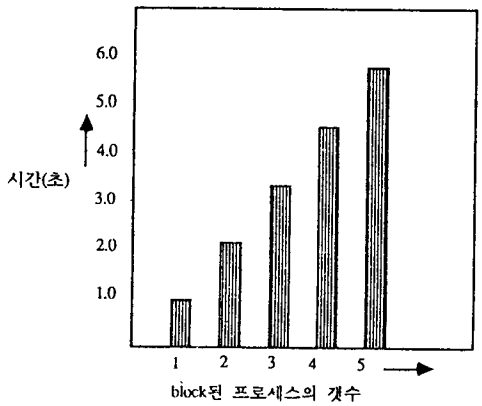


그림16. 일반(비 실시간) 프로세스의 응답시간
Fig. 16. The response time of the general process.

Ⅵ. 결　론

본 논문에서 제시한 실시간 자원 관리 기법을 시험을 위하여 구축한 실시간 운영체제에 적용한 결과는 시험 및 고찰에서 보는 바와 같이 한계수행시간에 기초한 실시간 처리를 위해 많이 개선됨을 볼 수 있다. 실시간 시스템은 자신이 의도한 모든 처리를 위해서는 이러한 기능 이외에 기본적으로 고성능 하드웨어의 지원이 있어야 할뿐 아니라, 고 신뢰성 특성을 갖는 하드웨어 및 소프트웨어의 특징을 고루 갖추어야 한다.

또한, 현재 실시간 시스템의 발전 경향은 분산시스템 환경하에서 각 노드는 다중 프로세스를 장착하고

있기 때문에 실시간 처리 기능을 지원하기 위한 기능들이 더욱 복잡한 구조를 지니면서 보다 고 차원적인 소프트웨어 기법들이 요구되고 있다. 이러한 환경에서 실시간 자원 관리 기법을 위해서 구조적인 보강이 있어야 하며 동시성을 만족하기 위한 기법들이 추가적으로 지원이 되어야 한다.

즉, 다중 프로세스들이 네트워크를 통해 연결된 분산 환경에서의 실시간 자원 관리를 위해 최적의 실시간 프로세스 스케쥴링 기법이 지원 되어야 할 뿐만 아니라, 노드들끼리의 실시간 통신 기법들의 지원이 절실히 요구되고 있다. 이러한 환경하에서 실시간 자원 관리 기법들이 추가적으로 연구 개발되어야 할 것이다.

參 考 文 獻

[1] B. Look and G. Ho, "Real-time extentions to the UNIX operation system," *Uniform conference Proceedings,* pp.293-299, Jan. 1984.

[2] D.R. Cheriton and M.A. Malcolm, "Thoth, a portable real-time operating system," Communications of the ACM, vol. 22, no. 2, pp. 105-115,

[3] Dan E. Ladermann and David J. Preston, "There is real timeliness in UNIX," Computer Design, pp. 105-115, Sep. 1983.

[4] H. Lycklama and D.L. Bayer, "The MERT operating system," The Bell System Technical Journal, vol. 57, no. 6, pp. 2049-2085, July-Aug. 1978.

[5] H.M. Deitel, An Introduction to Operating System, Chap 12-15, Addison Wesley, 1982.

[6] H. Tokuda, C.W. Mercer, "Priority inversions in real-time communication," Real-Time System Symposium, pp. 348-359, Decem. 1989.

[7] J.J. Wallace and W.W. Barnes, "Designing for ultrahigh availability: The unix RTR operating system," *IEEE computer,* pp. 31-39, Aug. 1984, Ferb. 1979.

[8] J.M. Scanlon, "The 3B20D processor & DMERT operating system," The Bell System Technical Journal, vol. 22, pp. 167-179, Jan. 1983.

[9] J.R. Kane, R.E. Anderson, and P.S. McCabe, "Overview, architecture and performance of DMERT," The Bell System Technical Journal, vol. 62, no. 1, pp. 181-189, Jan. 1983.

[10] J. Stankovic, "Real-time computing systems: the next generation," Tutorial Hard Real-Time Systems, pp. 14-38, 1988.

[11] M.J. Bach, The Design of the UNIX Operating System, Prentice-Hall Inc, 1986.

[12] P. Sherrod & S. Brenner, "Minicomputer system offers time sharing and real time tasks," *Computer Design,* pp. 223-228, Aug.1984.

[13] S. Cheng, T. Stankovic, "Scheduling algorithms for hard real-time systems: a brief survey," pp. 150-176, 1988.

[14] S.Evanczuk, "Real-time OS." Electonics, pp. 113-137, March 24, 1983.

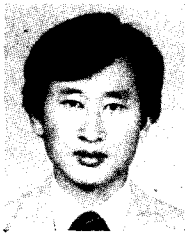
[15] W. Rauch-Hindin, "Real-time UNIX seizes new products, markets," *Mini-Micro Systems,* pp. 61-72, Sep. 1986.

[16] W.N. Toy & L.E. Gallaher, "The 3B20D processor & DMERT operting system," The Bell System Technical Journal, vol. 22, pp. 181-190, Jan. 1983.

[17] W. Zhao, K. Ramamritham, J. Stankovic, "Preemptive scheduling under thime and resource constraints," *IEEE Transactions on Computers,* pp. 24-35, Jan. 1987.

[18] J.B. Lee, H.J. Kim, K.W. Rim, Y.J. Park, "A study of about the implementation of real-time disk I/O scheduling on UNIX operating system," Australian Unix Users Group Conference, 1988. 9.

著 者 紹 介



李 楨 培(正會員)

1981年 경북대학교 공과대학 전자공학과 전자계산전공 졸업. 1983年 경북대학교 대학원 전자공학과 전자계산전공 석사과정 졸업. 1988年～현재 한양대학교 전자공학과 박사과정 재학중. 1982年～1991年 2月 한국전자통신연구소 컴퓨터연구단 선임연구원. 1991년 3월～현재 부산외국어대학 컴퓨터공학과 전임 강사. 주관심분야는 실시간 운영체계, 컴퓨터 네트워크 등임.

朴 容 震(正會員)

1969年 와세다대학 전자통신학과 졸업. 1971年 와세다대학원 석사 졸업, 일본 EDP(주) 근무. 1978年 와세다대학원에서 박사학위취득. 1979年～현재 한양대학교 전자공학과 교수. 1983年부터 1년간 Univ. of Illinois, Urbana 전산학과 방문교수. 주관심분야는 컴퓨터 네트워크, 분산시스템 등임.