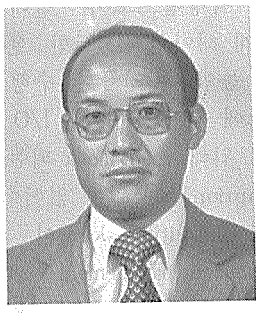
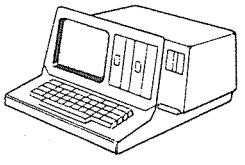


吳 吉 祿
韓國電子通信研究所
컴퓨터연구부장 / 工博

Real-Time UNIX Process Scheduling에 관한 연구



본 논문은 84년도 컴퓨터기술개발분야 국책연구과제중 "Real-Time용 미니컴퓨터개발" 내에서 Real-Time O.S.를 한국전자통신연구소의 시스템 소프트웨어연구실에서 수행한 Real-Time UNIX를 위한 Process scheduling의 연구에 관한 것으로서, UNIX의 multi-tasking 환경에서 실시한 프로세스가 외부로부터의 event에 대해 빠른 응답 처리를 가능하게 하기 위한 Preemptive process scheduler의 설계 및 구현과 그의 evaluation에 관해 기술한다.

1. 서 론

Real-time operating system을 설계하는데는 여러가지 고려사항이 있겠으나 응용 task들의 특성 및 요구사항을 얼마나 지지할 수 있느냐에 그 촛점이 놓여진다. 일반적으로 실시간 응용 task들은 주로 자료의 빠른 인식 및 처리를 요구하며, 이를 위해서는 processor의 처리 속도가 빨라야 함은 물론이지만 소프트웨어 즉, operating system측면에서는 실시간 task가 하드웨어 혹은 소프트웨어 인터럽트로부터의 빠른 응답을 가질 수 있도록 충분히 고려되어야 한다. 특히, UNIX와 같은 multi-tasking의 환경에서는 하나의 task가 인터럽트로부터 얼마나 빠른 응답을 가질 수 있느냐는 task-scheduling에 깊이 관여되는 문제이다. Task scheduler는 크게 preemptive와 non-preemptive의 두가지 형태가 있을 수 있는데, non-preemptive scheduler하에서 수행되고 있는 하나의 task는 더 높은 우선순위를 가진 task가 wake up 되더라도 processor의 control을 잃지 않는다. 즉, 인터럽트 처리때를 제외하고는 그 task가 스스로 자신의 수행을 중단하지 않는한(road-blocking) processor의 control을 내주지 않는다. 이러한 non-preemptive scheduling을 사용하는

시스템에서는 road-blocking간의 수행 시간을 제한함으로써 task들간의 상호 협조체제가 이루어질 필요가 있다. UNIX에서는 이를 위해 1초의 time-slice를 두고서 제한하고 있다. 이는 시간 문제가 크게 요구되지 않는 real-time 응용 task들에 대해서는 심각한 문제가 아니다. 한편 preemptive scheduler하에서 현재 수행되고 있는 하나의 tesk는 더 높은 우선 순위의 task가 interrupt handler에 의해 wake up될때 processor의 control을 빼앗기게 된다. 이는 interrupt handler에 의해 받아들여지는 자료가 그 handler에 의해 wake up되는 task에 의해 즉시 처리되어야 하는 그런 시스템에서는 매우 중요한 것이 된다. 자료가 전송되어 오는 속도가 앞의 자료가 처리되기전에 overwrite할수 있을지도 모르는 그런 정도라면 또한 preemptive scheduling이 필요하다.

이 논문에서는 Time Sharing Preemptive scheduling을 취하는 UNIX(V7)에서 real-time preemptive scheduler를 구현하고 평가한 것에 관해 기술한다.

2. UNIX Process Scheduler와 실시간

UNIX의 process scheduler는 time-sharing에 의한 multi-tasking을 구현하기 위해 CPU 사용을 근거로 한 우선 순위에 의해 preemptive scheduling을 하고 있다. 즉, READY process들 중에서 현재까지 CPU를 가장 적게 사용한 프로세스에게 scheduler는 CPU를 제공하게 되며, road-blocking에 의해 SLEEPING하고 있던 프로세스가 발생된 event에 의해 wake up되면 일반적으로 높은 우선 순위를 가지고서 낮은 우선 순위의 RUNNING 프로세스를 preemption하게 된다.

여기서 time-critical 실시간 프로세스가 이 시스템에서 수행되고 있다고 가정할 때, 이 프로세스가 wake up된 경우 항상 RUNNING 프로세스를 preemption하여 event에 대해 실시간 처리를 수행할 수 있다고 보장할 수 없을 뿐만 아니라 실시간 처리가 필요한 부분을 수해하는 도중에 time-slice로부터의 영향은 이러한 프로세스들이 의도하는 실시간 작업을 제대로 수행할 수 없게 만든다. 따라서 UNIX에서 실시간 프로세스를 time-sharing 환경으로부터 영향을 받지않고서 수행될 수 있게끔 scheduling 정책을 수립하는 것이 필요하다. 이를 위해서는 먼저 수행시간의 긴급성에 따라 부여되는 새로운 우선 순위가 결정되어야 하며 이 순위는 CPU 사용 시간에 무관하게 책정됨으로써 절대적인 preemption이 일어나게 해야한다. 또한 실시간 프로세스의 수행기간 동안 더 긴급한 수행시간을 요구하는 프로세스에 의하거나 혹은 I/O completion 등을 기다리기 위해 road-block 되는 일외에는 절대로 preemption되는 일이 없어야 하는 것이 이상적이겠으나 multi-tasking이 이루어지는 UNIX에서 resource management 상에서 실시간 프로세스가 본의 아니게 road-block되는 일이 있음을 고려할 때 이에 대한 최선의 해결책이 필요하다.

3. Real-Time Preemptive Scheduler의 설계 및 구현

Real-time process의 execution time requirement에 따른 preemptive scheduler는 앞에서 언급한 두가지 사실에 촛점을 두고 설계되었으며, 그 구현은 SSM-16/KONIX에서 이루어졌다.

1) Execution Level의 설정

수행시간의 긴급성에 따라, 기존의 time-sharing priority외에 또 하나의 우선 순위로서 execution level을 모든 프로세스에게 부여한다. 따라서 각 프로세스는 두 종류의 우선 순위 즉, time-sharing을 위한 우선순위와 real-time preemption을 위한 execution level을 가지게 된다. Scheduler는 RUNNABLE process들 중에서 항상 가장 높은 execution level의 process를 다음에 switch-in될 프로세스로 선정하며 그 level에서 RUNNABLE한 프로세스가 없으면 다음 낮은 level에서 선정한다. 같은 execution level를 가지는 프로세스간에는 기존의 UNIX scheduling 방법에 의해 time-sharing된다. 또한 더 높은 execution level의 프로세스는 더 낮은 execution level의 프로세스에 의해 절대로 preemption되지 않는다. 따라

서 하나의 실시간 프로세스가 preemption되는 경우는 다음의 세 경우 뿐이다.

가) 더 높은 execution level의 process가 wake up되거나 하여 RUNNABLE하게 되는 경우.

나) 같은 execution level에서의 다른 process가 RUNNABLE하여 time-sharing이 필요할때.

다) road-blocking.

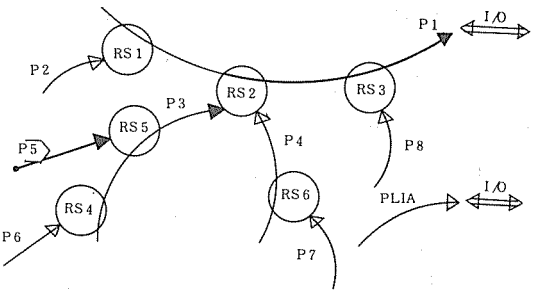
여기서 가), 나)의 경우는 예측된 event들이며 user에 의해 control될 수 있으나, 다)의 경우는 external I/O device로부터의 I/O completion을 기다리는 경우 외에 resource가 available하지 않음으로 인해 road block되어 본의 아니게 수행시간이 지연되는 경우가 있을 수 있는데 후자의 경우 real-time process가 얼마나 빨리 wake up될 수 있느냐는 전적으로 scheduling 정책에 달려 있다. 이는 UNIX의 reentrancy와는 무관하며 wake up된 후 switch-in될 때 까지의 지연시간은 non-reentrancy로 인해 발생되는 문제이다.

하나의 프로세스가 실시간 프로세스로 수행되게 하는 방법은 booting시에 결정되어 수행되게 하는 방법이 있을 수 있고 다른 하나 가능한 방법은 system call에 의한 것일 수 있는데 여기서는 후자의 방법을 채택하였는데 그 이유는 실시간 프로세스라고 해서 수행의 시작에서 끝까지 항상 critical time requirement를 갖는 것이 아닐 뿐더러 multi processing을 요구하는 경우에 각 프로세스의 time-requirement가 다를 수가 있어서, 따라서 수행시에 따라 혹은 프로세스에 따라 execution level이 제어되어야 할 경우가 많다고 고려되었기 때문이다.

Execution level은 크게 두 부분으로 나누어지는데 normal time-sharing process들에 대해서는 zero level이 주어지며 real-time process들에 대해서는 하나 이상의 양의 값의 level이 주어질 수 있다. 그 level들의 수는 그 system 내에서 수행될 real-time application의 성격에 따라 kernel generation시 tunning될 수 있다.

2) Road-block 시간을 최초로 하는 Execution Level의 Propagation

앞의 1)에서 road-block 되는 후자의 경우 가장 빨리 wake up될 수 있는 scheduling 정책을 여기서 제시한다.



RSX : resources

PX : processes

: request, road blocked

: real time process

그림 1 Execution Level Propagation

앞의 그림은 real-time process가 resource의 un-availabe로 인해 road-block이 되는 경우 가장 빨리 wake up될 수 있게 하기 위해 execution level이 전파되어지는 상황을 나타내고 있다.

실시간 프로세스 P5가 resource RS3의 un-available로 인해 road-block되고 앞의 그림과 같은 상황에서 I/O device로부터의 interrupt에 의해 P1과 P1'가 wake up되었을 경우 P1'보다는 P1이 수행하여 RS2가 release되고, P4보다는 P3가 run하여 RS5를 release함으로써 문제의 real-time process P5가 run되게 할 수 있다면, 이는 P5의 road-block 시간을 최소화 할 수 있는 유일한 path이다. 같은 resource에 대해 road-block되어 있는 모든 process들의 time-sharing priority는 같으며 P3 대신 P4가 switch-in되어 run될 경우, P4가 다시 road-block되거나 혹은 kernel mode를 빠져 나가게 될 때까지 P3가 RS5를 release하기 위해 run할 수 없으므로 그 동안 P5는 SLEEPING 상태로 있게 될 것이다. 따라서 P1, P3, P5의 순으로 schedule될 수 있게 하는 방법은 여러가지가 있을 수 있겠으나 여기서 가능한 하나의 방법을 제시한다.

가) 실시간 프로세스 P5가 road-block될 때 RS5가 P3에 의해 점유되어 있다는 사실을 알 수 있어야 하며, 이를 위해서 어떤 프로세스가 하나의 resource를 점유할 때마다 그 정보를 공고한다.

나) 가)의 방법으로 인해 P5가 P3를 찾았을 때 P3의 execution level을 P5만큼 일시적으로 높여주고, P3의 road-block 상태가 전과 같으면 마찬가지 방법으로 execution level을 전파시킨다.

여기서 원하는 scheduling은 이루어지지만 또 하나의 새로운 문제가 발생된다. 즉, P3가 RS5를 release한 후에 P5로부터 전파받은 execution level을 버리고 원래의 execution level을 가져야 하는데, 이때 P3가 점유하고 있었던 RS4와 그 이전의 resource를 요구하고 있는 프로세스가 P3의 원래의 execution level보다 높은 level을 가지고 있는 경우에 P3의 execution level을 원래 자신의 execution level이 아닌 또 다른 execution level로 천이되어야 한다. 이를 지지하기 위해 채택된 방법은 다음과 같다.

가) P3가 RS5를 release하게 될 때 자신이 점유하고 있는 모든 resource들에 대해 각 resource를 요구하고 있는 process들의 execution level들이 level별로 count되어 있어서 하나의 resource를 relrease할 때마다 그 resource의 excution level에 대한 count를 하나 줄이고 그 count가 zero인 경우에는 그 execution level로 천이하지 않고 count가 존재하는 다음의 낮은 level로 천이한다. 따라서 모든 resource들을 release한 후에는 원래 자신의 level로 되돌아 오게 된다. 이를 위해서 resource를 점유하게 되는 모든 프로세스들은 자신의 execution level transition table을 갖게되며 그 table의 내용은 P5와 같은 실시간 프로세스들에 의해서 채워진다.

나) P5는 road-block시 자신의 execution level을 P3의 transition table에 기록함으로써 P3에 의해 자신의 execution level을 전파시킬 수 있다.

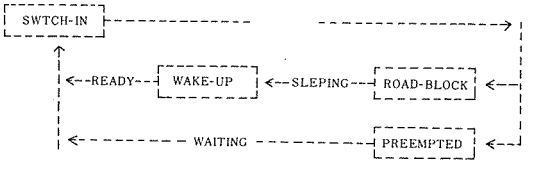


그림 2 Process Execution Cycle

4. Evaluation

여기서 구현된 real-time process scheduler의 evaluation을 위해 프로세스의 각 상태에 따라 수행시간을 추출하기 위한 하나의 UNIX system call이 구현되었다. 이 system call(tTr)은 그림-2에서 보이는 것처럼 프로세스가 수행되는 기간에서 갖는 네가지 상태에서 각각 소요된 시간을 1/1000초 단위로 측정한다.

Tracing 될 네가지 프로세스 상태(RUNNING, SLEEPING, READY, WAITING)의 천이를 일으키는 event는 switch-in, switch-out(roadblock 혹은 preempted), wake up에 의해서이다. 따라서 process status transition이 일어나는 이 세부분에서 완전한 time tracing이 가능하다. Trace에 필요한 buffer는 kernel space를 사용하며, 필요에 따라 allocate된다. 하나의 프로세스가trace되는 동안 각 상태에서의 소요된 시간과 횟수가 kernel buffer에 기록되었다가 trace가 끝나는 시점에서 그 동안 기록되었던 자료를 사용자가 제공한 buffer에 넣어준다.

이 evaluation system call의 user interface를 위해 일련의 C library가 만들어졌으며, 이 library는 사용자가 C program내에서 특정 routine들을 evaluate하기 위한 utility를 제공한다.

Evaluation을 위한 하나의 test program이 작성되었는데 이 test program은 같은 task를 수행하는 time-sharing process와 real-time process의 비교를 위해서 의도되었고 그 결과 real-time process는 time-sharing 환경으로부터의 영향을 거의 받지않고서 일정한 응답시간 및 수행시간을 가졌으며, 그 시간은 hardware system으로부터 받을 수 있는 최대의 시간에 해당하는 것이었다.

Test program은 test의 편의상 external event를 hard-disk로 사용하였다. 두개의 child process가 parent로부터의 start signal에 의해 거의 동시에 같은 task를 수행한다. 그 두개의 child process는 수행시작 전에 parent에 의해 서로 다른 execution level을 가지게 되는데, 하나는 time-sharing level 다른 하나는 real time level을 갖는다. Evalution에 사용된 task는 hard-disk에 존재하는 서로 다른 두 file system을 100K씩 read

만을 10번 수행하는데 이는 block I/O 를 위한 chash buffer에 이미 들어 있는 block을 읽는 경우를 피하기 위함이며 따라서 모든 read( ) 는 disk I/O를 실제 수행하게 된다. (현재 KONIX에서의 cache buffer는 60 blocks).

표 1

Exec. Lev.	0	1
Elapsed	22.700	11.405
status	sec. ms/4	sec. ms/N
READY	00.040/4	00.000/4
RUNNING	11.230/9	11.235/5
WAITING	11.300/4	00.000/0
SLEPING	00.130/4	00.170/4

결과(표 1)를 보면 child 1 과 child 2 가 거의 동시에 시작되었는데, child 2 (RT)는 일정한 elapsed time과 READY time(event로부터의 응답시간 : 1 clock 이하)을 가졌다. SLEPING time도 같은 결과를 보이나 이는 실제 hardware device로부터의 응답 시간으로서 software적으로 보장될 수 있는 시간은 아니다. Child 2 에 대한 이러한 결과는 더 많은 time-sharing task가 수행되고 있는 환경에서도 같은 결과이다. 따라서 결과적으로 이 real-time preemptive scheduler는 multi-tasking 환경에서 긴급한 수행시간을 요구하는 프로세스의 인터럽트 응답시간을 최소의 시간에 보장함을 알 수 있다.

5. 결론

이상의 내용으로부터 여기서 구현된 real-time process scheduler는 충분히 의도된 기능을 다함을 볼 수 있으며 여기서 얻어진 결과를 기초로 본격적인 real-time UNIX를 개발한다면 더 많은 응용분야를 지지할 수 있는 real-time operating system이 될 것으로 기대된다. 그러나 현재 이 scheduler는 UNIX 자체의 근본적인 문제와 함께 real-time을 위해 해결되어야 할 문제점이 몇가지 있다. 이러한 문제점에는 UNIX의 non-reentrancy와 preemptive scheduling을 위한 kernel overhead 등을 들 수 있는데, 이는 효율성이 좋은 IPC(Inter Process Communication)아래 kernel을 기능별로 modulization함으로써 해결될 수 있다. 하지만 이의 구현을 위해서는 많은 연구와 노력이 뒤따라야 할 것이다.

참고문헌

1. H. Lycklama and D. L. Bayer, "The MERT Operating System", The Bell System Technical Journal, Vol. 57, No. 6, July-August 1978

2. J. R. Kane, R. E. Anderson, and P. S. Mc-Cabe, "Overview, Architecture and Performance of DMERT", The Bell System Technical Journal, Vol. 62, No. 1, January 1983

3. Byron Look and Gary Ho, "Real-Time Extentions to the UNIX Operation System", Uniform Conferencc Proceedings, January 1984

4. Douglas J. Ross and M. Martin Taylor, "UNIX Support for Guaranteed Real-Time Processing", USENIX Summer Conference Proceeding, July 1983

5. Walter S. Heath, "A System Executive for Real-Time Microcomputer Programs", IEEE MICRO, June 1984

6. John J. Wallace and Walter W. Barnes, "Designing for Ultrahigh Availability : The Unix RTR Operating System", IEEE COMPUTER, August 1984

7. David R. Cheriton and Micael A. "Thoth, a Portable Real-Time Operating System", Communications of the ACM, Vol. 22, No. 2, February 1979

8. Dan E. Ladermann and David J. Preston, "There id Real Timeliness in UNIX", COMPUTER DESIGN, September 1983

9. Stephan Evanczuk, "Real-Time OS", Electronics, March 24, 1983