구조화된 예외 처리를 위한 Continuation 구현 기법

김세원,○ 김영필, 유혁
고려대학교
{swkim, ypkim, hxy}@os.korea.ac.kr

Continuation Implementation Method for Structured Exception Handling

Se-Won Kim,○ Young-Pill Kim, Hyuck Yoo
Korea University

요 약

Continuation은 운영체제의 실행 흐름을 최적화 하기 위한 수단으로 사용되어 왔다. 특히 Continuation을 이용하면 스택의 내용 없이, 특정 시점 이후의 프로세스의 연산을 그대로 수행 할 수 있는 장점을 가지고 있다. 하지만 이와 같이 스택에 저장된 정보를 사용하지 않을 경우, 구조화된 예외 처리 방법(SEH)을 사용하는 운영체제에서는 Continuation을 바로 적용하기에 어려움이 존재한다. 이러한 문제를 해결하기 위해, SEH를 위해 함수에서 수정한 스택의 내용을 저장해 두었다가 Continuation이 끝나고 스택을 복원하여 SEH가 올바르게 실행 되도록 하였다.

1. 서 론

Continuation은 연산의 나머지라는 의미로, 프로세스나 스레드의 실행의 특정 시점 이후의 할 일들에 대한 개념적인 정의를 의미한다. 운영체제에서 Continuation은 제어 흐름(control flow)의 최적화를 위한 수단으로 사용되어 왔다. 특히, Mach와 같은 마이크로 커널처럼 프로세스간 통신(Inter-Process Communication, IPC)의 성능이 운영체제의 성능을 좌우하는 경우, Continuation을 통해 이를 극복하려는 시도가 있었다.

마이크로 커널의 경우 커널 서비스를 구현한 프로세스들 간 통신이 발생하며, 상대 프로세스의 응답을 기다려야 하는 상황이 발생한다. 따라서 프로세스 실행 중 상대 프로세스의 응답을 기다리기 위해서는 block이 발생하게 된다. Continuation은 프로세스간 통신시 발생하는 block을 최적화하는 기법이다[1].

본 논문은 마이크로소프트의 Windows Research Kernel (WRK)의 wait 시스템 콜을 Continuation으로 구현하는데 발생한 문제를 해결한 방법에 대해 설명한다. WRK는 교육 및 연구용으로 마이크로소프트의 Windows 운영체제의 핵심 부분의 코드를 공개한 커널이다.

Windows의 모든 시스템 콜은 시스템 콜 처리시 발생하는 예외를 처리하기 위해 구조화된 예외 처리(Structured Exception Handling, SEH)방식을 사용한다. 즉, 모든 시스템 콜은 실행 중 발생 할 수 있는 예외를 try-except 문을 이용하여 시스템 콜 랩퍼함수(Wrapper Function)에 전달 할 수 있도록 한다. Continuation을 이용하여 wait 시스템 콜을 구현할 때의 문제는, *Continuation이 스택의 정보를 이용하지 않고, block이후에 연산을 수행하기 때문에, SEH에 전달할 예외 값이 올바르지 않는다*는 문제가 있었다.

따라서, 다음 두 가지 조건을 만족해야만, Windows의 wait 시스템 콜을 Continuation으로 구현 할 수 있었다. 첫 번째는, Continuation을 이용하여 wait 시스템 콜이 block이후에도 스택의 정보 없이 진행을 할 수 있어야 한다. 두 번째는, SEH의 동작을 방해하지 않으면서, wait 시스템 콜을 Continuation으로 구현해야 한다. 이 조건을 만족하기 위해서, 스택의 함수 호출 프레임을 분리하여 Continuation함수 복귀 과정에 필요한 스택 프레임을 재구성 하는 방법을 사용했다.

본 논문은 다음과 같이 구성된다. 2장은 배경 지식으로 Continuation, WRK의 wait 시스템 콜 동작, 구조화된 예외 처리에 대해서 설명한다. 3장은 wait 시스템 콜을 Continuation으로 구현한 방법에 대해서 설명하며, 4장은 수정한 wait 시스템 콜의 동작에 대해서 설명한다.

2. 배경 지식

2.1 Continuation

Continuation은 앞서 설명한 정의와 같이 특정 시점 이후의 프로세스가 할 일들에 대한 개념적인 정의를 의미한다. 이것은 본래 scheme과 같은 언어에서 first-class Continuation에서 유래한 것이다. 하지만, [1]에서

Continuation을 운영체제의 최적화에 응용을 시작으로, 다른 연구에서도 Continuation을 이용한 최적화에 사용되기 시작되었다.

일반적으로 현재 사용되는 컴퓨터 구조에서의 실행 흐름은 함수 호출의 연속으로 이루어 진다. 그리고 연속된 함수 호출은 일련의 복귀(return)을 통해서 자신(callee)을 호출한 호출자(caller)로 되돌아가 실행

결과를 전달한다. 이때 함수의 호출과 복귀를 구현하기 위한 자료구조로 스택이 사용된다. 스택의 후입선출(Last In First Out)의 특성은 함수를 호출할 때 현재 상태를 저장하고 단순한 포인터 이동으로 다음 함수 호출로 넘어 갈 수 있도록 한다.

Continuation은 스택을 이용한 함수 호출과 달리 스택을 사용하지 않는 특징이 있다. 복귀를 통해 호출자로 돌아가는 대신, 복귀 이후 호출자가 해야 할 일을 다시 연속된 함수 호출로 진행한다. 복귀가 필요없기 때문에 Continuation은 스택을 필요로 하지 않는다.

2.2 Windows의 wait 시스템 콜의 동작

Windows의 시스템 콜은 3개의 함수를 거쳐 동작한다. 그림 1은 Windows에서 wait 시스템 콜이 동작할 때의 함수 호출 관계이다. 유저 프로세스에서 시스템 콜을 호출하면, 커널의 시스템 콜 엔트리에서 호출한 시스템 콜의 번호를 이용하여 프로세스에서 요청한 시스템 콜 함수를 호출한다. 시스템 콜 엔트리를 통해 커널에 진입하면, executive layer의 함수는 넘겨받은 핸들러를 커널 내 유효한 포인터로 변환하고 실제 시스템 콜의 역할은 KeWaitForSingleObject 함수에서 수행한다.

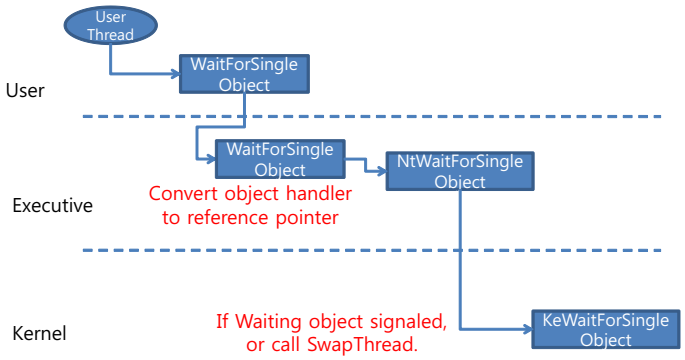

그림 1 Windows의 Wait 시스템콜 동작 흐름

이때, Executive layer의 NtWaitForSingleObject 함수는 KeWaitForSingleObject를 호출할 때, SEH를 이용하여 KeWaitForSingleObject가 실행하면서 발생한 예외를 처리한다.

2,3 구조화된 예외 처리(SEH)

Windows의 시스템 콜은 try, except를 통해 구조화된 예외 처리를 위한 방법을 제공한다. 구조화된 예외 처리는 발생할 수 있는 예외를 처리 할 수 있는 핸들러를 등록하고 예외가 발생 하였을 시 등록한 핸들러를 호출함으로써 발생한 예외를 처리 할 수 있도록 한다.

구조화된 예외 처리는 스택을 이용한 방법이 주로 사용되며 스택을 통해 예외가 발생했을 때의 CPU의 레지스터 정보를 예외 핸들러에게 전달하여 처리하도록 한다. SEH를 이용한 함수를 disassemble하여 분석해 보면, 함수의 시작 부분에 SEH preamble이라 하여 처리하고자 하는 예외 핸들러를 등록한다. 그리고 try와 except 사이에서 예외를 처리하고자 하는 함수를 호출하여 예외 발생시 미리 등록한 함수의 예외 핸들러를 호출하도록 되어 있다.

```
NTSTATUS
NtWaitForSingleObject (
    __in HANDLE Handle,
    __in BOOLEAN Alertable,
    __in_opt PLARGE_INTEGER Timeout
    )
{

        try {
            Status = KeWaitForSingleObject(...);

        } except((GetExceptionCode () == STATUS_MUTANT_LIMIT_EXCEEDED)?
                    EXCEPTION_EXECUTE_HANDLER :
                    EXCEPTION_CONTINUE_SEARCH) {

            Status = GetExceptionCode();
        }

        ObDereferenceObject(Object);


    return Status;

} ? end NtWaitForSingleObject ?
```

그림 2 구조화된 예외 처리의 사용 예

그림 2는 구조화된 예외 처리를 사용한 예이며, KeWaitForSingleObject 실행 시 발생한 예외를 처리한다. 일반적인 함수 호출 방식에서는 이와 같은 구조화된 예외 처리 방식은 동작하는데 있어서 문제가 없었다. 하지만, KeWaitForSingleObject함수를 Continuation으로 구현할 경우 구조화된 예외 처리가 올바르게 동작하지 않는 문제가 발생하고 이로 인해 시스템 콜을 요청한 사용자 프로세스가 올바르게 동작하지 않는 문제가 발생하였다.

Continuation으로 구현한 함수로 인해 구조화된 예외 처리가 문제가 되는 이유는 Continuation이 이전에 저장해 놓은 스택의 정보를 사용하지 않고 진행하기 때문인데 그로 인해 이전에 등록해 놓은 SEH의 핸들러 정보들이 손실되어 예상하지 못한 결과를 가져오게 되었다.

3. Wait 시스템 콜의 Continuation으로의 구현

이번 장에서는 Windows의 wait 시스템 콜을 Continuation으로 구현에 대해 살펴 볼 것이다. 3.1

절에서는 Wait 시스템 콜을 Continuation을 다시 구현하기 위해 사용한 함수 분리(function split)에 대해 살펴 보고 3.2절에서는 분리한 함수의 동작을 살펴 본다. 3.3절에서는 앞서 제기된 Continuation을 통해 시스템 콜을 구현할 경우 SEH에서 발생되는 문제를 해결하기 위한 방법을 제시한다.

3.1 Continuation 구현을 위한 wait 함수 분리

특정 함수를 Continuation형태로 구현하기 위해 사용되는 방법으로 함수를 특정 지점-이 경우 block이 되는 지점이 된다-을 기준으로 post-blocking과 pre-blocking 부분으로 나눈다. 이렇게 함수를 분리를 한 후, blocking 이후의 프로세스가 실행할 부분은 post-blocking부분이 되며 이것을 별도의 함수로 다시 만들어 blocking 이후에 호출 하는 형태로 구현한다. Windows의 wait 시스템 콜은 단순하게 block되는 시점을 기준으로 post, pre-blocking 부분으로 나누기가 어려운 문제가 있다. Wait 시스템 콜은 무한 룹(Infinite loop)을 돌면서 대기하는 object의 가용 여부에 동작이 바뀐다.

그림 3은 Windows의 wait 시스템 콜의 형태를 간략하게 표현한 것이다. 여기에서 대상이 되는 함수는 KeWaitForSingleObject이다. 무한 룹으로 인하여 최종적으로 나뉘는 부분은 총 4개가 된다. 그리고 함수 내에서 호출되는 KiSwapThread() 함수는 문맥전환(context switch)함수로써 block이 되는 시점이 된다. 이와 같이 Continuation으로 구현하려는 함수가 무한 룹을 포함하는 경우 4 부분으로 나누어 구현하는 방법은 [2]에서 제시되었다.

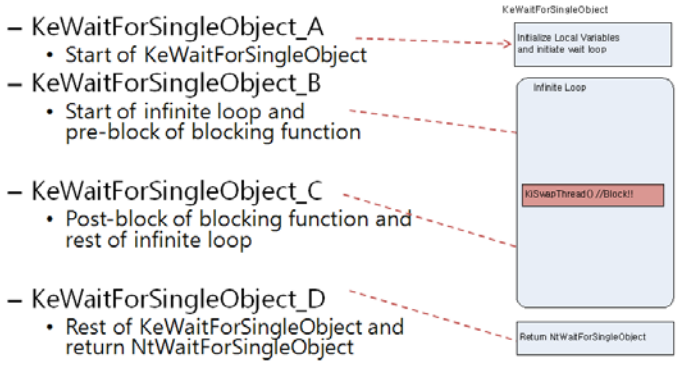

그림 3 KeWaitForSingleObject 함수의 분리

3.2 분리한 wait 시스템 콜의 동작

KeWaitForSingleObject 함수를 4부분으로 나누면, 이 함수들의 동작 순서를 잘 정의해 주어야만, 원래의 함수와 동일하게 동작하게 된다. 그림 4는 wait 시스템 콜을 4개의 함수로 나눈 후 각 함수들의 동작 순서의 관계를 나타낸 그림이다. 그림에서 원은 분리된 함수를 의미하고 원과 원 사이에 간선은 함수 호출 관계를 나타낸다. 이 그림에서 간선 3과 간선 4는 원래 KeWaitForSingleObject 함수에 있던 무한 룹을 의미하며, 간선 2와 간선 4는 block이후 실행되는 Continuation을 의미한다.

Continuation을 사용하지 않을 경우, block이후 wait 시스템 콜은 스택내의 정보와 문맥 전환 시 저장되었던 CPU의 상태 정보를 통해서 block이후의 연산을 의도한대로 실행 할 수 있다. 반대로 Continuation을 이용하여 구현할 경우, block이후 KeWaitForSingleObject_C 함수를 호출하는 형태로 block이후의 과정을 똑같이 재현해 낼 수 있다. 하지만 이와 같은 분리 만으로는 Continuation을 구현 할 수는 없다. 왜냐하면, block되기 전의 함수의 지역 변수의 값들을 알 수 없기 때문에 이 값들을 block되기 전에 저장하고 block이후에 복원하는 과정이 부가적으로 필요하게 된다. 이와 같은 지역변수는 함수 실행 시 변화되는 정보들이며, 보통 실행 컨텍스트라 불린다. 지역 변수는 함수가 호출될 때, 스택에 지역변수를 위한 공간이 할당된다. Continuation을 구현하기 위해서 이러한 실행 컨텍스트 정보는 thread의 TCB(Thread Control Block)에 저장하여 스택과 별도로 관리되도록 하였다. 이와 같은 수정을 통해서 Continuation함수는 스택의 내용과는 무관하게 동작이 가능하고 결국 block이후 C함수를 호출 할 때는 stack의 정보 없이 실행이 가능하다.

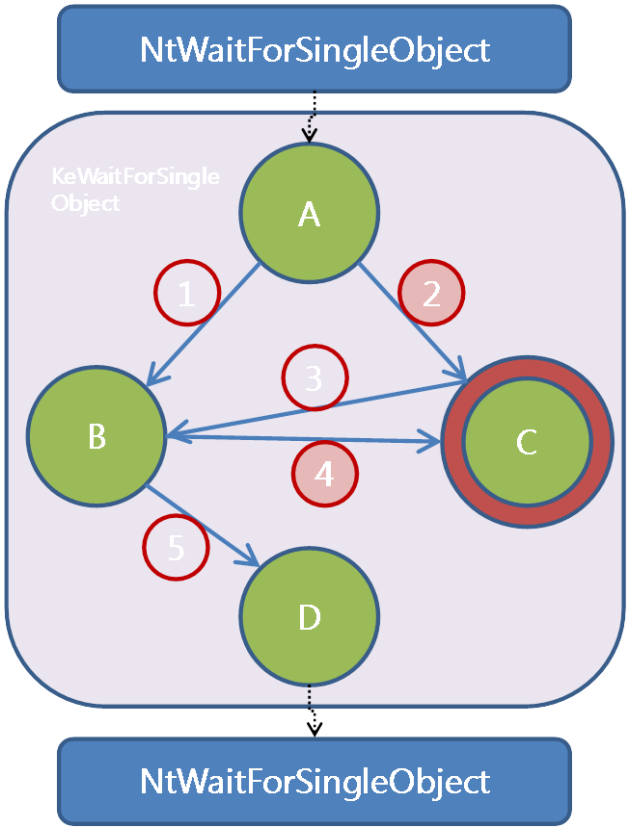

그림 4 4개로 분리한 wait 시스템 콜의 동작 다이어그램

하지만 2.3절에서 언급한 바와 같이 Continuation을 통해 stack의 정보를 사용하지 않는 구조로 변경하면, 구조화된 예외 처리 방식을 사용하는 Windows구조상 발생하는 예외를 처리하지 못하는 문제가 발생하였고, 이를 해결하기 위해서 별도의 스택 정보를 관리해야 하는 부가적인 작업이 필요하게 되었다.

3.3. Stackframe의 저장과 복원

Stackframe이란 함수를 호출할 때, 호출한 함수에서 사용하는 지역 변수, 넘겨받은 인자 그리고 호출자의 복귀 주소를 저장한 공간이다. 이러한 내용들은 스택에 저장되는데, 각각의 함수 호출마다 Intel 프로세서의 경우 EBP레지스터를 이용하여 현재 실행중인 함수의 stackframe을 저장한다[4].

그림 5는 wait 시스템 콜을 호출 하였을 때의 call stack정보이다.

```
nt!KeWaitForSingleObject_A
nt!NtWaitForSingleObject+0xe8
nt!KiFastCallEntry+0xfc
WARNING: Stack unwind information not available.
ntdll+0x2ed54
0x77e6ba42
0x758433f9
0x77e6608b
```

그림 5 Wait 시스템 콜의 call stack정보

현재 실행중인 함수는 KeWaitForSingleObject_A이며 이 함수를 호출하기 위해서 KiFastCallEntry를 시작으로 NtWaitForSingleObject 함수를 거쳐왔다. 여기에서 문제는 block이후 KeWaitForSingleObject_C를 호출할 때 이와 같은 스택 정보를 무시하게 되므로 NtWaitForSingleObject함수에 설치된 SEH를 무시하는 결과가 초래하게 된다.

이러한 문제를 해결하기 위해서 block함수인 KiSwapThread를 호출하기 전에 KiFastCallEntry부터 NtWaitForSingleObject함수의 Stackframe을 별도로 저장한 후에, KeWaitForSingleObject_C를 호출하기 전 스택에 복원하는 방법을 사용하였다. 그 결과 SEH의 동작은 보장하면서 wait시스템 콜은 Continuation을 사용하여 동작하는 결과를 얻을 수 있었다.

4. Evaluation

이번 장에서는 Continuation으로 구현한 wait 시스템 콜의 동작이 올바르게 실행되는 것을 보인다. Evaluation을 위해 스택을 그대로 두고 wait 시스템 콜을 실행하도록 했다.

```
nt!KeWaitForSingleObject_C+0x5
nt!KiSwapThread+0x353
nt!KeWaitForSingleObject_B+0x2ea
nt!KeWaitForSingleObject_C+0x175
nt!KeWaitForSingleObject_A+0x63
nt!NtWaitForSingleObject+0xe8
nt!KiFastCallEntry+0xfc
WARNING: Stack unwind information not available. Following frames may be
ntdll+0x2ed54
0x77e6ba42
nt!DbgkCreateThread+0x3ac
0x670027ca
```

그림 6 Block이후 KeWaitForSingleObject_C의 실행

그림 6은 block함수인 KiSwapThread이후 KeWaitForSingleObject_C가 실행 중인 상황의 call stack정보이다. 그림 4에서 보인 4개의 함수 동작 다이어그램에 따라서 함수가 실행됨을 확인 할 수 있다. KeWaitForSingleObject_C에서 더 이상 wait을 할 필요가 없을 경우 시스템 콜을 마치고 user 프로세스로 복귀를 해야 하는데, 이때 NtWaitForSingleObject_Finalize라는 함수가 3.3 절에서 설명한 Stackframe 복원을 수행한다. 그림 7은 NtWaitForSingleObject_Finalize에 진입했을 때의 call stack 정보이며, 그림 8과 그림 9에서 stackframe을 복원하여 원래 되돌아가야 할 함수 정보들로 call stack 정보들이 복원됨을 보이고 있다.

```
nt!NtWaitForSingleobject_Finalize
nt!KeWaitForSingleObject_C+0x2f
nt!KiSwapThread+0x353
nt!KeWaitForSingleObject_B+0x2ea
nt!KeWaitForSingleObject_C+0x175
nt!KeWaitForSingleObject_A+0x63
nt!NtWaitForSingleObject+0xe8
nt!KiFastCallEntry+0xfc
WARNING: Stack unwind information not available. Following frames may be
ntdll+0x2ed54
0x77e6ba42
nt!DbgkCreateThread+0x3ac
0x670027ca
```

그림 7 NtWaitForSingleObject_Finalize 실행

```
nt!NtWaitForSingleobject_Finalize+0x1c
nt!NtWaitForSingleObject+0xe8
nt!KiSwapThread+0x353
nt!KeWaitForSingleObject_B+0x2ea
nt!KeWaitForSingleObject_C+0x175
nt!KeWaitForSingleObject_A+0x63
nt!NtWaitForSingleObject+0xe8
nt!KiFastCallEntry+0xfc
WARNING: Stack unwind information not available. Following frames may be
ntdll+0x2ed54
0x77e6ba42
nt!DbgkCreateThread+0x3ac
0x670027ca
0x77e6608b
```

그림 8 Stackframe 복원과정 - 첫번째 frame인 NtWaitForSingleObject 복원

```
nt!NtWaitForSingleObject+0xe8
nt!KiFastCallEntry+0xfc
WARNING: Stack unwind information not available. Following frames may be
ntdll+0x2ed54
0x77e6ba42
nt!DbgkCreateThread+0x3ac
0x670027ca
0x77e6608b
```

그림 9 Stackframe 복원과정 - 두번째 frame인 KiFastCallEntry 복원

이와 같은 과정을 통해서 wait 시스템 콜을 Continuation 으로 구현하더라도 기존에 사용중인 구조화된 예외 처리를 사용하는 Windows 시스템 콜 구현 방식을 지킬 수 있었다.

4. 관련 연구

Continuation 을 운영체제의 실행 흐름에 최적화에 이용한 연구는 오래 전부터 진행되어 왔다. 그 중 Richard P. Draves 가 Mach 3.0 의 IPC 를 개선한 연구를 시작으로 많은 연구가 있었다[1]. Mach 는 대표적인 마이크로 커널으로 커널 서비스를 사용하기 위해 잦은 IPC 로 인해 성능이 저하되는 문제가 발생하였고, 대부분의 경우 상대 프로세스에게 메시지를 전송하고 그것을 받는 과정에서 오버헤드가 발생하였다. 이를 해결하기 위해 Continuation 을 이용하여 IPC 를 참여하는 두 프로세스의 커널 스택을 공유하도록 하여 기존의 IPC 성능을 향상 시켰다.

Continuation 이 운영체제의 커널 스택 경량화에 기여하는 점에 초점을 두어 커널 스택을 1 개만을 사용하는 연구가 진행되었다. Continuation 을 이용하면, 커널영역을 실행하는 프로세스의 커널 스택이 block 이후에 유지할 필요가 없음을 착안하여 L4 커널을 1 개의 커널 스택만으로 동작하는 형태로 설계하였다[3].

여러 개의 프로세스들이 협업을 하는 프로그래밍 모델에서도 Continuation 이 사용되었다[5]. 이 연구에서는 I/O 를 통해 프로세스 실행중 block 이 되는 이후의 과정을 Continuation 으로 구현하였다. Continuation 을 이용하기 위해서는 스택을 일일이 관리를 해주어야 하는 단점이 발생하는데, 이를 자동적으로 처리할 수 있는 프로세스 협업 모델을 제안하였다.

5. 결 론

본 논문은 Continuation 을 이용하여 마이크로소프트사의 Windows Research Kernel 의 wait 시스템 콜 구현하고, 이로 인해 발생한 SEH 의 문제점을 해결하였다.

SEH 는 스택에 예외 처리 핸들러의 주소를 저장하기 때문에 Continuation 형태로 시스템 콜을 구현하여 스택의 정보를 사용하지 않을 경우 등록된 핸들러를 사용할 수 없게 된다. 이를 위해서 wait 시스템 콜을 실행하고 복귀하기 전에 stackframe 의 내용을 저장하고 복원하는 부가적인 과정을 추가하였으며, SEH 의 올바른 동작을 확인 할 수 있었다.

참고 문헌

[1] Richard P. Draves, Brain N. Bershad, Richard F. Rashid, and Randall W. Dean, “Using Continuations to Implement Thread Management and Communication in Operating Systems" ACM SIGOPS Operating Systems Review, Volume25, Issue 5. Pages:122-135. Oct. 1991

[2] Richard P. Draves, “Control Transfer in Operating System Kernels”, Ph. D. Dissertation, School of Computer Science, Carnegie Mellon University, 1994

[3] Matthew Warton, “Single Kernel Stack L4”, BS. Dissertation, University of New South Wales, 2005

[4] Intel 64 and IA-32 Architecture Software Developer’s Manual Volume 1, http://www.intel.com/products/processor/manuals/

[5] Atual Adya, Jon Howell, Marvin Theimer, William J. Bolosky, John R. Douceur, “Cooperative Task Management without Manual Stack Management”, 2002 Usenix Annual Technological Conference, June 2002.