CHILL 언어의 병행처리를 위한 Run-time 지원 시스템의 설계 및 구현

하 수 철† · 조 철 희††

요 약

본 논문은 ITU-T 통신 처리 시스템용 프로그래밍 언어 CHILL에서 제공되는 병행처리 기능을 적용하기 위한 CHILL 실행시간 지원 시스템(CRS: CHILL Run-time support System)의 설계 및 구현에 관한 연구이다. CHILL은 다른 병행 프로그래밍 언어에 비해 다양한 병행처리 기능들을 제공하고 있기 때문에, CRS의 설계는 병행처리를 위한 주요 기능들과 기법들을 획득할 수 있는 효과를 얻을 수 있다. 본 논문에서는 CHILL 컴파일러와의 정합을 위한 병행처리 기능의 인터페이스 규격을 설계한다. CHILL의 병행처리 프리미티브는 프로시듀어 호출 형식으로 사용하도록 라이브러리 방식을 사용하며, CHILL 프로세스의 실행을 병행적으로 제어하기 위해 CHILL 프로그램 구동 루틴 및 문맥 교환부와 CHILL 프로세스 제어부를 구현한다.

A Design and Implementation of Run-time Support System for Concurrent Processing of the CHILL

Soo-Cheol Ha† · Cheol-Hye Cho††

ABSTRACT

This paper presents a design and implementation of CRS(CHILL Run-time support System) to adapt the concurrent processing facilities of CHILL(CCITT High Level Language) which had recommended by ITU-T(International Telecommunication Union Telecommunication Standardization Sector). Because the CHILL provides more various concurrent processing facilities than other concurrent programming language, a design and implementation on CRS can give us real effects to gain the major functionalities and the techniques of the concurrent processing. In this paper, we design the interface rules of concurrent functions to conform with the CHILL compiler. We use the concurrent processing primitives as the library style to be invoked by procedure calls, and implement the start-up routine of the CHILL program, the context switching routine, and the CHILL process control parts to control the execution of the CHILL processes concurrently.

1. 서 론

병행처리 프로그래밍에 대한 연구는 크게 두 가지로 구분할 수 있다. 하나는 기존의 순차 언어로 작성된 프로그램을 병렬로 수행할 수 있는 프로그램으로 변환시키는 병렬화 컴파일러에 대한 연구[1,2]이며, 다른 하나는 프로그래머가 직접적으로 병행 프로그램을 작성하도록 병행처리 구조들을 지원하는 병행처리 프로그래밍 언어 및 환경에 대한 연구이다. 전자의 경우 프로그램의 병행 실행을 위해 필요한 모든 조치를 컴파일러가 처리하므로 이상적인 방법으로 보이지만 실제

† 종신회원 : 대전대학교 컴퓨터공학과 교수
†† 정 회 원 : 한국전자통신연구원 책임연구원
논문접수 : 1998년 9월 14일, 심사완료 : 1999년 3월 29일

로 병렬화 컴파일러에 의해 처리되는 부분이 극히 제한적이고, 데이터 종속성과 같은 복잡한 문제를 해결해야 하기 때문에 효율성이 떨어진다. 반면에, 후자는 프로세스간 통신 및 동기화와 같은 모든 병행처리와 관련된 사항을 프로그래머가 직접 설계 작성해야 한다는 부담이 따르지만 프로그래머가 직접 병행처리 프로그램을 설계 및 작성할 수 있는 장점이 있다. 이 접근법에 의해 Concurrent Pascal[3], Concurrent C[4], Modular-2[5], CHILL[6], OCCAM[7] 및 ADA[8] 등과 같은 다양한 병행처리 프로그래밍 언어들이 개발되었다.

이러한 병행처리 프로그래밍 언어로 작성된 병행 프로그램은 다중 프로세서 시스템과 단일 프로세서 시스템에서 모두 실행 가능해야 하는데, 이는 별도의 지원 도구에 의한 실행 환경을 요구한다. 즉, 이러한 실행 환경은 일반적으로 병행처리 언어에 대한 컴파일러와 함께 제공되지만, 새로운 병행 프로그래밍 언어를 개발하거나 이미 설계된 병행 프로그래밍 언어의 컴파일러를 개발할 경우에는 병행처리를 위한 실행 환경의 개발이 추가로 요구된다는 의미이다.

본 논문에서는 CHILL 병행처리 언어로 작성된 병행 프로그램에 대한 병행처리 실행 환경을 제공하기 위한 CHILL 실행시간 지원 시스템(CRS : CHILL Run-time support System) 시스템 설계 및 구현에 대하여 기술한다. 이 시스템은 ITU-T(International Telecommunication Union - Telecommunication Standardization Sector)에서 통신 시스템의 소프트웨어 개발에 사용할 목적으로 정의하여 권고한 병행 프로그래밍 언어인 CHILL(CCITT High Level Language)을 대상으로 한다. 그러나 최근에 연구 개발되고 있는 객체지향 CHILL[17,18,19]은 본 연구 범위에 포함되지 않는다.

2. CHILL 언어의 병행처리 기능

2.1 개 요

ITU-T 권고안 Z.200[6]에 의해 공표 된 CHILL 언어는 저장 프로그램 제어 방식(Stored Program Control System)을 사용하는 전자식 교환기의 소프트웨어 개발에 사용할 목적으로 개발한 언어로서, 유사한 다른 분야의 통신 시스템 소프트웨어 개발은 물론 범용 프로그램 작성에도 사용할 수 있는 프로그래밍 언어이다.

CHILL 언어는 강력한 모듈 개념의 언어로서 병행처리를 위한 프로세스 정의, 생성 및 소멸 등과 같은 프로세스 관리 기능을 비롯하여 프로세스간의 동기화를 위한 EVENT 구조, 공유 자원에 대한 상호배제 기능을 지원하는 REGION 구조, 그리고 프로세스간의 정보 전달을 위한 통신 기능을 지원하는 SIGNAL 및 BUFFER 구조 등의 기능들을 제공한다.

2.2 CHILL 프로세스[9,10]

CHILL 언어는 프로세스 개념에 의한 병행성을 제공한다. 따라서, CHILL 프로세스는 CHILL 언어에서 제공하는 병행성의 기본 단위로서 특정 기능을 수행하기 위해 순차적으로 수행될 실행문들의 집합으로 정의되며, 이러한 프로세스들은 동시에 병행적으로 실행될 수 있다.

CHILL 언어의 동기화 기능들을 사용하여 작성된 CHILL 프로세스는 다른 프로세스들과 동기화 되고 통신하며 실행되지만, 동기화 기능들을 사용하지 않고 작성된 프로세스는 다른 프로세스의 실행과 무관하게 독립적으로 실행된다.

2.2.1 프로세스 정의

CHILL 프로세스의 정의는 프로세스가 사용할 변수들에 대한 기억 장소 및 프로시듀어들과 같은 지역적인 객체들의 선언과 수행할 동작에 대한 일련의 실행문들을 포함한다. 하나의 프로세스 정의에 대해 여러 개의 프로세스가 생성될 수 있으며, 생성된 각 프로세스는 동일한 동작을 수행한다.

2.2.2 프로세스 생성 및 소멸

현재 수행 상태에 있는 전체의 CHILL 프로그램을 가상의 Outermost 프로세스라 하며 시스템에 의해 생성된다. 그러나, CHILL 프로그램에서 프로세스 정의문에 의해 정의된 각 프로세스는 실행중인 Outermost 프로세스나 이미 생성된 또 다른 프로세스가 START 실행문을 수행하여 생성되고, 실행중인 프로세스가 STOP 실행문을 수행하여 프로세스의 실행이 종료되고 프로세스가 소멸된다. 일반 프로시듀어에게 매개변수를 전달하는 것과 동일하게 프로세스 생성시 생성될 프로세스에게도 매개변수를 전달할 수 있다. 그러나 프로세스의 수행 결과로 어떤 값을 반환할 수는 없다.

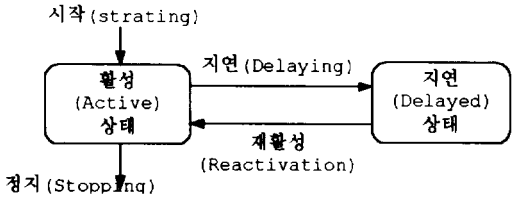
2.2.3 프로세스 식별

각각의 CHILL 프로세스는 프로세스 고유의 인스턴스 값에 의해 식별되며, 프로세스 인스턴스는 프로세

스 생성시에 유일한 값으로 할당된다. 아울러 각 프로세스는 실행 도중에 THIS 연산자에 의해서 자신에 대한 프로세스 인스턴스 값을 얻을 수 있으며, 상대 프로세스의 인스턴스는 병행처리 관련 명령어 구문의 SET절에 의해 얻을 수 있다.

2.3 CHILL 프로세스의 상태[6,10]

CHILL 프로세스의 상태는 현재 실행 중인 "활성(active)" 상태와 어떤 조건이 만족될 때까지 수행이 일시적으로 정지된 "지연(delayed)" 상태로 구분된다. 프로세스가 활성 상태에서 지연 상태로 천이 되는 것을 "지연"이라 하며, 지연 상태에서 활성 상태로 천이되는 것을 "재활성(reactivation)"이라 한다. CHILL 프로세스의 기본적인 상태 천이는 (그림 1)과 같다.



(그림 1) CHILL 프로세스의 기본 상태 천이도

특히, 이미 다른 프로세스가 점유하여 사용 중인 REGION에 대한 점유 요구에 의해 일시적으로 그 수행이 정지된 프로세스도 CHILL에서는 활성 상태로 간주한다.

2.4 CHILL 언어의 병행처리 구조[6,10]

2.4.1 EVENT 구조

CHILL의 EVENT 구조는 프로세스간 동기화를 위한 기능을 제공한다. CHILL 프로세스들은 EVENT 변수 및 관련 명령문을 사용함으로써 프로세스간 동기 문제를 명시적으로 구현할 수 있다. 해당 EVENT에 대해 지연된 프로세스의 대기 큐로 간주되는 EVENT 변수는 일반 변수와 유사한 방법으로 선언되며, EVENT 선언에는 다음과 같은 내용이 포함된다.

- EVENT의 기억 장소 이름
- 해당 EVENT에 동시에 대기할 수 있는 프로세스의 수를 나타내는 큐길이

EVENT을 이용하여 프로세스간의 동기화를 구현할 수 있는 실행문은 DELAY, DELAY CASE, 그리고 CONTINUE가 있으며, 각각의 역할은 다음과 같다.

- DELAY : 프로세스를 특정 EVENT가 발생할 때까지 지연시킴.
- DELAY CASE : 동시에 다수의 EVENT에 대하여 프로세스를 지연시킴.
- CONTINUE : 지정된 EVENT을 발생시킴.

프로세스가 DELAY나 DELAY CASE 실행문을 수행할 때 이미 주어진 EVENT의 길이의 수 만큼의 프로세스가 지연되어 있는 경우 그 프로세스는 지연되지 않으며 "DELAYFAIL" 예외상황(exception)을 발생시킨다.

2.4.2 BUFFER 구조

BUFFER 구조는 공유 메모리를 사용한 프로세스간의 동기화 및 통신 기능을 제공하며, BUFFER에 대한 실행문에는 BUFFER 메시지 송신에 사용되는 SEND와 메시지 수신에 사용되는 RECEIVE CASE 문 및 RECEIVE 문이 있다.

각 BUFFER는 송신된 BUFFER 메시지를 보관할 공유 메모리를 가지고 있으며, BUFFER 선언문에는 BUFFER에 대한 공유 메모리 영역을 정의하는 다음과 같은 내용이 선언되어야 한다.

- BUFFER 메시지에 대한 모드(데이터 형)
- 공유 메모리의 크기(BUFFER 길이)

BUFFER에 대한 메시지 송수신 동작은 다음과 같은 절차에 따라 수행된다.

(1) BUFFER 메시지 송신시

- BUFFER에 저장된 메시지의 개수가 BUFFER 길이보다 작은 경우 메시지를 보내려고 하는 프로세스는 BUFFER 메시지를 BUFFER 장소에 저장하고 수행을 계속함.
- BUFFER 메시지가 BUFFER 길이 만큼 저장되어 있을 경우 메시지를 보내려는 프로세스는 수행이 지연됨.

(2) BUFFER 메시지 수신시

- 해당 BUFFER에 저장되어 있는 메시지가 있는 경우 그 메시지를 수신한 뒤 수행을 계속함.
- 수신할 메시지가 없는 경우 해당 프로세스는 지연됨.

2.4.3 SIGNAL 구조

SIGNAL은 메시지 전송에 의한 프로세스간의 동기화 및 통신을 위한 기능을 제공한다. 각 SIGNAL은 다음과 같은 내용을 정의문에 포함시켜야 한다.

- SIGNAL 메시지에 대한 데이터 모드
- SIGNAL 전송을 위한 통신 채널 관련 사항(시그널을 받을 상대 프로세스 식별에 관한 사항)

SIGNAL에 대한 실행문은 SEND 및 RECEIVE CASE가 있으며, SEND 실행문은 비블록킹(non-blocking) 전송 방식을 채택하고 있다. 그러나 SIGNAL을 수신하는 RECEIVE CASE 실행문은 프로세스가 SIGNAL을 수신하지 못했을 경우 해당 SIGNAL이 수신될 때까지 프로세스의 수행을 일시적으로 지연시키는 블록킹 수신 방식을 사용한다.

SIGNAL 전송을 위한 통신 채널은 프로세스 정의명 및 인스턴스를 모두 사용할 수 있다. 프로세스 정의명을 사용하여 SIGNAL을 송신한 경우 SIGNAL 메시지가 해당 프로세스 정의명에 의해 생성된 모든 프로세스 인스턴스에게 전달되는 브로드캐스팅(broadcasting) 개념이 적용된다. 이러한 경우 수신 프로세스 인스턴스에 대한 프로세스 정의가 틀리면 SENDFAIL 예외가 발생하며, 인스턴스가 존재하지 않는 프로세스에게 SIGNAL을 송신하는 경우에는 "EXTINCT", 또는 "EMPTY" 예외상황이 발생한다.

2.4.4 REGION 구조

REGION 구조는 REGION으로 선언된 영역 내에 선언된 변수들에 대한 기억 장소나 프로시듀어 등과 같은 데이터 객체들에 대한 상호 배타적인 접근(mutual exclusive access)을 위한 기능 및 구조를 제공한다.

REGION은 매우 강력하게 제한된 가시성을 가지고 있기 때문에 REGION 영역 내에서 선언된 데이터 객체들은 외부로부터 임의적인 접근을 원칙적으로 불허하며, 단지 REGION 영역 내에서 정의되고 그 REGION 밖으로부터 접근이 허용된 임계 프로시듀어(critical procedure)를 통해서만 가능하다.

또한, REGION 영역은 임의의 순간에 하나의 프로세스에 의해서만 점유될 수 있기 때문에 이미 다른 프로세스가 점유하여 사용하고 있는 REGION 영역에 접근하기 위해 해당 임계 프로시듀어를 호출할 경우 호출 프로세스는 수행이 일시적으로 중지된 대기 상태로 된다. 이러한 프로세스는 REGION 영역을 점유하여 사용하던 프로세스가 해당 REGION 영역의 사용을 종료하였을 때 해당 REGION에 대한 사용 권한을 받게 되며 중지되었던 수행을 계속하게 된다.

2.5 CHILL 언어의 병행처리 기능 분석

일반적으로 병행처리를 위하여 요구되는 요소 기능으로는 ① 프로세스 관리기능, ② 프로세스간 동기화 기능, ③ 프로세스간 통신 기능, ④ 공유자원 접근에 대한 상호배제 기능 등이 있다[9].

이러한 병행처리 요소 기능에 대하여 CHILL에서 제공하는 기능은 <표 1> 같이 요약 분류할 수 있다.

〈표 1〉 CHILL의 병행처리 기능 요약

구 분	역 할
1) 프로세스 관리 기능	
- 프로세스 정의	-프로세스의 정의
- START	-프로세스의 생성 및 구동
- STOP	-프로세스의 실행 종료 및 소멸
- THIS	-프로세스 식별자(인스탄스) 획득
2) 프로세스간 동기화 기능	
- EVENT	-프로세스 동기화용 공유변수 정의
- DELAY	-해당 EVENT가 발생할 때까지 수행지연
- DELAY CASE	-동시에 다수의 EVENT 들이 발생할 때까지 수행 지연
- CONTINUE	-정의 된 EVENT를 발생시킴
3) 프로세스간 통신 기능	
- BUFFER	-통신을 위한 공유 메모리 정의
- SIGNAL	-메시지 전달을 위한 SIGNAL 정의
- SEND	-BUFFER 또는 SIGNAL 메시지 전송 요구
- RECEIVE	-단일 BUFFER 메시지 수신 요구
- RECEIVE CASE	-다수의 BUFFER 또는 SIGNAL 메시지 수신 요구
4) 상호배제 기능	
- REGION	-모니터(monitor) 개념 제공 -임계영역 설정 -상호배제 기능 제공

큐 형태로 관리되는 EVENT는 가산 세마포어(counting semaphore)와 유사한 개념을 제공하므로, EVENT 길이를 가산 세마포어의 계수 값으로 설정하면 가산 세마포어가 요구되는 응용에 사용할 수 있다. 또한 BUFFER는 공통 메모리 영역을 사용하여 메시지를 전송하기 때문에 동일 프로세서 내에서나 공통 메모리를 사용하는 다중 프로세서 시스템에서의 메시지 전송에 적합하며, SIGNAL은 직접 프로세스에게 메시지를 전송하기 때문에 공통 메모리를 사용하지 않는

다중 프로세서 시스템에서의 프로세스간의 메시지 전송에 적합한 구조이다. 특히, BUFFER 길이가 0인 경우는 BUFFER 메시지를 저장할 메모리가 없는 경우로서 메시지를 전송하려는 프로세스는 수신 대기 프로세스가 없으면 항상 지연되며, 반대로 수신 프로세스도 전송 대기 프로세스가 없으면 항상 대기 상태로 되기 때문에 동기화 구현에 이용할 수 있다.

아울러 CHILL 언어의 REGION 구조는 공유 자원에 대한 동기화와 상호배제를 실현할 수 있는 Monitor 개념과 동일하며, 임계영역의 정의 및 접근 제어에 대한 제어 기법을 프로그래머가 직접 정의하고 구현할 수 있는 수단을 제공한다.

3. CHILL 병행처리 시스템의 구현 사례 분석

NTT의 CCOS(Concurrent CHILL Operating System)[11]은 운영체계와 응용 프로그램간의 명확한 인터페이스를 제공하고 병행처리, 실시간 처리, 물리적 입출력 처리 수단들의 지원을 목표로 설계되었다. CCOS는 병행처리 기능들을 가진 기초적인 운영체계로서 CHILL 언어로 작성된 다수의 실시간 응용 프로그램의 실행을 지원한다. CCOS는 CHILL 프로그램의 병행 수행을 지원하기 위한 효과적인 실행 지원 루틴들을 제공하며, 이러한 루틴들은 프로세스 생성 및 소멸, 프로세스 스케줄링, 프로세스간 통신 등을 수행한다. 아울러 입출력 처리, 실시간 처리 및 하드웨어 종속적 동작 등 CHILL 언어로서 기술할 수 없는 기능들에 대해서는 별도의 운영체계 프리미티브들을 제공한다. CCOS는 다음과 같은 5개의 계층 구조와 기능을 갖는다.

계층1) 가장 내부의 커널 계층으로 타이머 인터럽트 핸들러, 메모리 관리 프로그램, 그리고 스케줄러를 포함한 프로세스 관리 프로그램, 즉 CHILL 병행처리 기능을 위한 실행 지원 루틴 등과 같이 모든 응용에 공통적으로 적용되는 루틴들로 구성된다.

계층2) 하드웨어 이벤트에 대한 응답, 시스템 부팅 및 장애 처리 및 복구 절차 등과 같은 실시간 및 하드웨어 종속적 동작들을 수행하는 프로세스들로 구성된다.

계층3) 시간 종속적인 처리를 수행하며, 응용 프로그램의 요구에 따라 입출력 처리를 수행한다.

계층4) 운영체계와 CHILL 인터페이스를 제공한다. 인터페이스 기능은 CHILL의 병행처리 기능과 CHILL 프로시듀어들에 의해 정의된 운영체계 프리미티브들의 집합체로 구성된다. 프리미티브들은 CHILL 프로시듀어와 동일한 형태로 제공되기 때문에 응용 프로그램들은 CHILL에 새로운 인터페이스 수단의 추가 없이 곧 바로 사용할 수 있다.

계층5) 가장 밖에 있는 다섯 번째 계층은 CHILL로 작성된 응용 프로그램이다.

Philips A/G COS(CHILL Operating System)[12]는 VAX/VMS 상에서 개발된 것으로 CHILL 언어 정의에 의해서 요구된 모든 병행처리를 위한 실행시간 지원 기능들을 제공하며, CHILL 컴파일러의 코드 생성기와의 인터페이스를 실현한다. COS는 기계 종속적인 부분과 기계 독립적인 부분으로 구성된다. 기계 독립적인 부분은 CHILL 언어 정의에 의해 규정된 모든 서비스들을 제공하며, CHILL 컴파일러의 코드 생성기와 인터페이스 된다. CHILL 컴파일러와의 인터페이스는 모드(데이터 형) 및 프로시듀어로 정의되며, 응용 프로그램과 COS와의 인터페이스는 프로시듀어 호출에 의해 실현된다. 이 부분을 구성하는 프로세스 관리부, 입출력 관리부), 메모리 관리부 및 실행시간 지원부에 대한 기능적 역할은 다음과 같다.

1) 프로세스 관리부 : 프로세스 인스턴스 관리, 프로세스 동작 감시, 프로세스 스케줄링 등 CHILL의 병행처리 기능을 위해 필요한 모든 기능들을 지원한다.
2) 입출력 관리부 : 동시에 실행되는 다수의 프로세스로부터의 입출력 요구를 처리한다. 모든 입출력은 비동기적으로 제어된다. 따라서, 임의의 시간에는 하나의 프로세스만이 입출력을 진행할 수 있으며, 입출력을 요구하는 다른 프로세스들은 입출력 대기 큐에 대기된다.
3) 메모리 관리부 : Stack, Heap, Pool의 3가지 메모리 구성에 대한 할당, 해제 및 관리 기능을 지원한다.
4) 기본 Run Time 지원부 : Powerset, Bit-string, Character-string, String-slice, Array-slice에 대한 연산과 같이 복잡한 연산식에 기본 프로시듀어들과 CHILL의 동적 조건에 대한 실행 시간 검사 등에 대한 기본적 프로시듀어를 제공한다.

CSELT의 Concurrency Handler[13,14]는 CHILL 언어를 기본 프로그래밍 언어로 하고 UNIX를 기본 운영

체계로 하는 분산 프로그래밍 환경에서 CHILL 프로그램의 병행 실행을 지원하는 소규모의 네트워크 운영체제이다. 이러한 개발 환경에 대한 소프트웨어 구조는 4계층의 구조인 호스트 운영체계, 프로그래밍 지원 환경, 인터페이스, 그리고 지원 도구 및 응용 프로그램들로 구성된다. 계층1인 커널 부분은 호스트 운영체제로서 전체 환경을 지원하기 위해 필요한 최소의 기능 집합으로서 분산 환경에서 커널과 CHILL에 의해 제공되는 기능 및 수단들을 설정하고 가능하게 만든다. 프로그래밍 지원 환경인 계층2는 분산 환경을 위한 병행 처리기를 구성하는 병행성 제어 루틴들로 구성된다. 계층3은 최소한의 인터페이스 루틴들로서 프로그램들이 독립적으로 수행될 수 있도록 하기 위한 병행성 서버 역할을 수행한다. 이러한 구조는 분산 환경과 비 분산 환경 모두에서 프로그램간 통신을 위한 기법을 제공할 수 있으며, 메시지 형식의 통신 기법을 사용한다. 계층4는 개발 지원 도구 및 CHILL로 작성된 응용 프로그램이다.

또한 병행 처리기는 분산 환경에서 CHILL의 병행처리 기능들을 사용하기 위하여 설계되어, 다음과 같은 처리 기능을 수행한다.

1) 프로세스의 시작과 종료 관리
2) 프로세스간 동기화 및 통신
3) SIGNAL, EVENT 및 CHILL 프로세스들에 대한 정보의 유지 관리
4) 작업 노드 상에서 활성화될 다음 프로세스의 결정
5) 다른 노드에서 생성된 SIGNAL과 EVENT들의 전달 및 수신처리

CHILL 프로그램의 병행처리를 위한 실행시간 시스템의 구현 사례를 분석한 결과, CHILL 컴파일러 및 호스트 시스템과의 인터페이스, 프로세스 관리, 메모리 등의 자원관리, 입출력 처리 등이 CHILL의 병행처리를 위해 제공되어야 함을 알 수 있다. 본 논문에서는 이러한 구현 사례에 대한 기능 분석 내용을 기초로 프로세스 제어를 위한 문맥저장 및 복원과 같은 기계 종속적인 부분에 대한 구현과 CHILL 컴파일러와의 인터페이스 부분의 설계 및 구현에 대해 수행한 바를 논한다.

4. 병행처리 CHILL 실행시간 지원 시스템 설계

4.1 개요 및 설계 목표

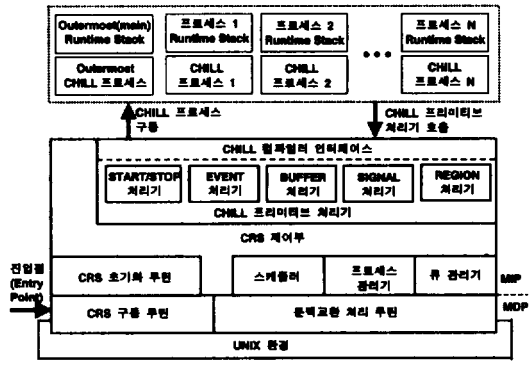
CHILL 언어의 병행처리 기능은 어떠한 하드웨어나 운영체제의 지원도 요구하지 않는다. 따라서 CHILL 언어의 병행처리 기능은 개발자가 구현 방법을 자유롭게 택할 수 있다[10]. 한편 프로그래밍 언어 측면에서 병행성을 지원하는 방법으로는 기존의 프로그래밍 언어에 명시적인 병행 언어 구조를 추가하는 방법과 라이브러리 형태로 지원하는 방법이 있다[15,16]. 3장의 구현 사례 분석결과로 얻어진 구현 대상에 대해 병행처리 기능을 라이브러리 형태로 구현한다.

CHILL 실행시간 지원 시스템(CRS : CHILL Run-time support Sytem)을 개발하기 위해 설정한 설계 목표는 다음과 같다.

1) CHILL 언어에서 제공하는 병행처리 기능에 대한 의미를 명확히 처리하도록 한다.
2) 이식성을 최대로 확보하기 위하여 기계 종속적인 부분을 최소화하여 설계한다.
3) 기능 추가 및 삭제, 기능 변경, 그리고 유지 보수를 고려하여 모듈화 개념을 사용하여 설계한다.
4) CHILL 컴파일러 및 호스트 시스템과의 인터페이스가 용이한 구조로 설계한다.

4.2 CRS의 구조

CRS는 크게 기계 종속적인 부분(MDP : Machine Dependent Part)과 기계 독립적인 부분(MIP : Machine Independent Part)으로 구성하였다. MDP는 CRS의 구동과 CHILL 프로세스의 문맥교환 루틴으로 구성하며, MIP는 CRS 초기화, CRS 제어, 스케줄링, 프로세스 및 큐 관리와 관련된 루틴, 그리고 CHILL 언어의 병행처리 기능들에 대한 프리미티브 루틴들로 구성한다. (그림 2)는 설계된 CRS의 시스템 구조를 보여준다.



(그림 2) CHILL 실행시간 지원 시스템(CRS)의 구조

1) CRS 구동 루틴 : 병행 CHILL 프로그램이 CRS의 제어 하에 수행되도록 하는 CRS 진입점(entry point)을 제공한다. CRS 구동 루틴은 CRS 상태를 초기화하기 위하여 CRS 초기화 루틴을 호출한다.
2) CRS 초기화 루틴 : CRS가 사용하는 각종 변수 및 큐들에 대한 초기 설정을 수행하고, Outermost 프로세스를 생성한다.
3) CRS 제어부 : CRS의 커널부로서 다음에 활성화될 프로세스를 선택하기 위하여 스케줄러를 호출하며, 생성된 모든 CHILL 프로세스의 실행 상태를 감시하여 모든 프로세스의 실행이 종료되면 전체 프로그램의 실행을 종료시키기 위해 제어를 CRS 구동 루틴으로 반환시킨다.
4) 프로세스 및 큐 관리기 : 해당 CHILL 프로세스들이 생성되고, 스케줄링 되어 수행되는 상황을 연결 리스트 형태의 큐를 이용하여 관리한다.
5) 문맥교환 처리 루틴 : 프로세스의 실행 상태인 문맥을 저장하는 루틴과 회복시켜주는 루틴으로 구성되며, 현재 수행중인 프로세스가 대기 상태가 될 때 해당 프로세스의 레지스터 문맥을 해당 프로세스 제어 블록(PCB : Process Control Block)의 문맥 보관 영역에 저장하거나, 스케줄링 되어 수행될 프로세스의 문맥을 PCB의 문맥 보관 영역으로부터 복구한다.
6) CHILL 프리미티브 처리기 : CHILL에서 지원하고 있는 각 병행처리 실행문에 해당하는 기능들을 처리하며, CHILL 컴파일러와의 인터페이스를 제공한다. CRS의 각 CHILL 프리미티브 처리기는 다음과 같다.
   ① START/STOP 처리기 : CHILL 프로세스의 생성시 프로세스의 PCB를 초기화하고 프로세스가 사용할 스택을 배정하며, 프로세스에게 전달될 파라미터를 처리한다. 또한 프로세스의 실행 종료시 프로세스 소멸 처리와 관련된 동작들에 대한 처리를 제어한다.
   ② EVENT 처리기 : EVENT과 관련된 EVENT 대기 큐의 초기화 및 관리, EVENT 식별자 부여, EVENT에 대한 프로세스의 지연 및 재 활성화 등에 대한 처리를 제어한다.
   ③ BUFFER 처리기 : BUFFER와 관련된 공유 메모리의 초기화 및 관리, BUFFER 식별자 부여, BUFFER 메시지 저장 및 전달, BUFFER 대기 큐의 관리, 그리고 BUFFER에 대한 프로세스의 지연 및 재 활성화 등에 대한 처리를 제어한다.
   ④ SIGNAL 처리기 : SIGNAL와 관련된 SIGNAL 대기 큐의 초기화 및 관리 제어, 프로세스간 SIGNAL 송수신, SIGNAL에 대한 프로세스의 지연 및 재 활성화 등에 대한 처리를 제어한다.
   ⑤ REGION 처리기 : REGION과 관련된 큐의 초기화 및 REGION 식별자를 부여하며, REGION에 대한 프로세스의 점유 요구에 대한 제어를 수행한다. 또한 REGION 사용을 위해 대기중인 프로세스와 REGION내에서 발생하는 프로세스의 지연 및 재 활성화 등에 대한 처리를 제어한다.

4.3 CRS의 구성 요소간 상호 동작

CRS를 구성하는 각 모듈들의 구동 절차 및 상호 동작 알고리즘은 다음과 같다.

[단계1] CHILL 프로그램의 실행이 요구되면 진입점인 CRS 구동 루틴의 실행이 시작되어 CRS 초기화 루틴을 호출한다.

[단계2] CRS 초기화 루틴은 CRS의 실행 및 CHILL 프로세스의 병행적 실행을 위해 필요한 메모리 등을 할당하고 프로세스 관리에 필요한 각종 큐들을 초기화 한 후, CRS 커널의 실행이 필요한 시기에 CRS 커널로의 복귀를 위하여 CRS 커널모드에 대한 문맥을 저장하고, CHILL의 주(main) 모듈을 실행시키기 위한 가상의 outermost CHILL 프로세스를 생성시키고 스케줄러를 호출한다.

[단계3] 스케줄러는 outermost 프로세스를 실행을 스케줄링 한다.

[단계4] 가상 프로세스인 outermost 프로세스는 CHILL의 주 모듈을 구동한다. 주 모듈에 기술된 START 명령어에 대한 CHILL 프리미티브 핸들러의 START/STOP 프리미티브가 호출되면, 해당 프로세스에 대한 PCB를 할당하고 실행대기 큐에 삽입한다.

[단계5] 실행대기 큐에 대기중인 CHILL 프로세스들이 스케줄러에 의해 스케줄링 되면 프로그램 코드가 순차적으로 실행된다.

[단계6] 프로세스의 실행 코드가 순차적으로 실행되는 과정에서 프로그래머에 의해 프로그램 된 CHILL의 병행처리 명령어를 만나면 해당되는 CHILL 프리미티브를 호출하게 되며, 호출된 프리미티브는 해당되는 병행처리 과정을 수행하게 된다.

[단계7] 각각의 CHILL 프로세스가 CRS의 병행처리 루틴을 실행하면서 실행을 지속할 수 없는 상태가 되면 문맥 교환 루틴을 호출하여 자신의 문맥을 저장하고, 제어를 스케줄러에게 넘겨 실행대기 상태에 있는 프로세스가 스케줄링 될 수 있도록 한다.

5. 병행처리 CHILL Run-Time 시스템 구현

5.1 CHILL 컴파일러와의 인터페이스

CRS와 CHILL 프로그램과의 기본적인 인터페이스 구조는 프로시듀어 호출 형식으로 정의하여 설계하였다. 즉, CRS는 CHILL 언어가 제공하는 병행처리에 필요한 기능들을 프로시듀어 형태의 프리미티브 루틴으로 구현하여 병행처리 지원 라이브러리로 구성하여 제공한다. CHILL 컴파일러는 CHILL 프로그램 번역시 CHILL 언어의 병행처리와 관련된 데이터 선언문들에 대하여 메모리 영역을 할당하고 초기화한다. CHILL

〈표 2〉 CHILL 병행처리 프리미티브 변환 규칙

프리미티브	CHILL 구문	CRS 인터페이스 명세
START	START processName([ParameterList]) [SET instanceLoc] [OS Directives];	start(Addr,procDef,stackSize, priority,timeLimit, systemOrUser,typeOfStart, interval,unitOfInterval, tartTime, actCount, numOfArgs [,actualParamList]);
STOP	STOP;	stop();
THIS	THIS;	this();
DELAY	DELAY EVENTLoc [PRIORITY priority];	delay(->EVENTLoc, priority);
DELAY CASE	DELAY CASE [{SET instanceLoc [PRIORITY priority]; | PRIORITY priority;}] (EVENT_1): action_statement_list; . . . (EVENT_n): ction_statement_list; ESAC;	delaycase(->instanceLoc, priority, numOfEVENT, ->EVENT_1, ..., ->EVENT_n);
CONTINUE	CONTINUE EVENTLoc;	continue(->EVENTLoc);
BUFFER SEND	SEND bufLoc(bufMsgLoc) [PRIORITY priority];	sendbuf(->bufLoc,priority, ->bufMsgLoc);
BUFFER RECEIVE	BufMsgLoc:= RECEIVE bufLoc;	rcvbuf(->bufLoc,->bufMsgLoc);
BUFFER RECEIVE CASE	RECEIVE CASE [SET instanceLoc;] (bufLoc_1 [IN msgLoc_1]): action_statement_list . . . (bufLoc_n [IN msgLoc_n]): action_statement_list [ELSE action_statement_list] ESAC;	rcvbufcase(cmdType, ->instanceLoc, numOfBUFFER, ->bufLoc_1, ->msgLoc_1, . . . ->bufLoc_n, ->msgLoc_n);
SIGNAL SEND	SEND sigName (sigMsgLoc) [TO instance] [PRIORITY priority];	sendsig(SIGNALId, procDefId, destination,priority, sizeOfValue,->sigMsg);
SIGNAL RECEIVE CASE	RECEIVE CASE [NONPERSISTENCE [EXCEPT save_SIGNAL_list]] [SET instanceLoc;] (sigName_1 [IN msgLoc_1]): action_statement_list . . . (sigName_n [IN msgLoc_n]): action_statement_list [ELSE action_statement_list] ESAC;	rcvbufcase(cmdType,->instanceLoc, numOfSIGNAL, numOfSaveSIGNAL, SIGNALId_1, ->msgLoc_1, . . . SIGNALId_n, ->msgLoc_n, saveSIGNALId_1, . . . saveSIGNALId_n);
REGION	REGIONName : REGION <REGION body> END;	enterREGION(->REGION) <REGION body> exitREGION(->REGION)

프로그램에 기술된 병행처리 실행문들에 대해서는 각각 대응되는 CRS의 CHILL 프리미티브 루틴에 대한 프로시듀어 호출로 변환한다. 해당 병행처리 기능을 제어하기 위하여 CRS가 필요로 하는 일련의 정보들을 산출하여 파라미터로 전달할 수 있는 형식으로 변환하여 호출문과 파라미터 전달을 위한 코드를 생성한다.

이러한 과정을 걸쳐 생성된 CHILL 프로그램의 목적코드와 병행처리 지원 라이브러리를 링크하여 최종적인 실행 모듈을 산출한다.

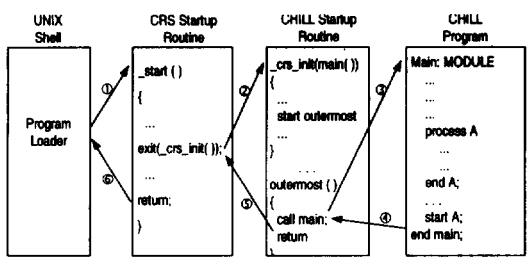
앞의 <표 2>는 CHILL 컴파일러가 CHILL 프로그램의 병행처리에 대한 실행구문을 CRS의 CHILL 프리미티브 루틴에 대한 호출문으로 변환하기 위해 설계한 변환 규칙이다.

5.2 호스트 시스템과의 인터페이스

CHILL 프로세스들의 병행적 실행을 제어하기 위해서 CHILL 프로세스에 대한 모든 제어가 CRS에 의해서 이루어져야 한다. 따라서 CHILL 프로그램에 대한 실행이 요구되면 CRS가 최초로 실행되도록 하여 CRS가 CHILL 프로그램에 정의된 프로세스들의 생성 및 실행을 제어하도록 호스트 시스템과의 인터페이스를 설계하였다.

CRS의 호스트 시스템의 운영체제인 UNIX에서는 프로그램의 실행을 구동하는 방법으로 시스템의 전역변수인 "_start"를 이용한다. 즉, 모든 프로그램에 대한 실행 요구에 대하여 운영체계에 의해 항상 "_start"라는 프로그램 구동 루틴으로 제어를 넘겨주는 것으로 시작되며, 이 루틴에서 사용자 프로그램의 "main" 루틴을 호출함으로써 사용자가 작성한 프로그램의 실행이 구동 되도록 하는 방법을 사용한다.

CRS의 구동 체계도 이와 유사한 방법으로 설계하였다. 즉, 운영체계의 프로그램 구동 루틴에서 "main"을 호출하는 대신에 CHILL 프로그램의 수행에 대한 제어 권한을 가지고 있는 CRS를 호출하도록 변경함으로써 CRS가 실행되도록 할 수 있으며, 이 때부터는 모든 CHILL 프로세스의 구동 및 실행에 대한 제어를 CRS가 처리한다. (그림 3)은 이러한 인터페이스 과정을 도식화한 것이다.



(그림 3) 병행 CHILL 프로그램의 구동 과정

이 CRS의 Vax-11 계열 구동 루틴은 (그림 4)에 보인다.

```
char **environ = (char **)0;

asm("#define _start start");
asm("#define _eprol eprol");
extern    unsigned char          etext;
extern    unsigned char          eprol;

start()
{
  struct kframe {
    int    kargc;
    char  *kargv[1];    /* 크기는 kargc에 의존 */
    char  kargstr[1];   /* 변동 크기 */
    char  kenvstr[1];   /* 변동 크기 */
};
  register int r11;              /* init를 위해 필요 */
  register struct kframe *kfp;  /* r10 */
  register char **targv;
  register char **argv;

  asm("    movl       sp,r10");

  for(argv=targv= &kfp->kargv[0];*targv++; )
   ;
  if (targv >= (char **)(*argv))
    --targv;
    environ = targv;
asm("eprol:");
/* "main" 대신에 "_crs_init" 호출 */
exit(_crs_init(kfp->kargc, argv, environ));
}
asm("#undef _start");
asm("#undef _eprol");
```

(그림 4) CRS의 구동루틴

5.3 프로세스 제어 블록(PCB)

PCB은 프로세스에 대한 실행 정보를 보관, 관리하기 위한 것으로서 CRS는 CHILL 프로세스를 생성할 때 다음과 같은 정보들이 저장하거나 처리하는 PCB를 생성하여 프로세스를 관리한다.

- 프로세스의 정의 식별자
- 프로세스의 인스턴스
- 프로세스의 상태
- 우선순위
- 실행시간 스택의 기준 주소(base address)
- 병행처리 기능 관리 정보(EVENT, BUFFER, SIGNAL, REGION에 관한 정보)
- 프로세스의 문맥 정보(레지스터 값)

PCB에 대한 데이터 구조는 다음과 같다.

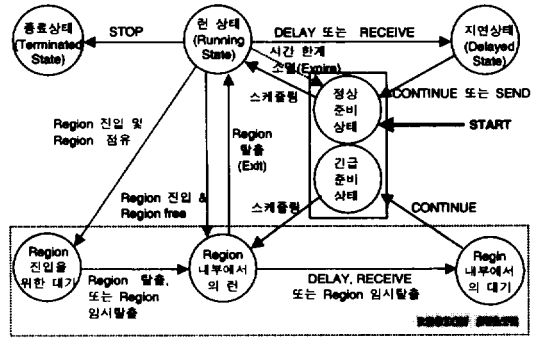
```
struct processControlBlock {
  PROC_DEF_MODE procDefId;    /* 프로세스 정의 ID */
```

```
  INSTANCE_MODE instance;     /* 프로세스 인스탄스 */
  unsigned state;             /* 현 상태 */
  PRIORITY_MODE runPriority;  /* 런 우선순위 */
  PRIORITY_MODE queuePriority; /* 큐 우선순위 */
  unsigned long stackBase;    /* 스택의 기준 주소 */
  unsigned long stackTop;     /* 스택의 톱 주소 */
  int       sizeOfStack;      /* 스택 크기 */
  DCL_EVENT_NODE *pDclEventList;
                              /* 선언된 EVENT 리스트 */
  DELAY_CASE_MODE *pDelayCase;
                              /* EVENT Delay-Case 포인터 */
  DCL_BUF_NODE *pDclBufList;
                              /* 선언된 BUFFER 리스트 */
  BUF_RCASE_MODE *pBufRecvCase;
                              /* BUFFER Recv-Case 포인터 */
  unsigned *pBufMsg;   /* 수신된 BUFFER 메시지 */
  int       regionState;  /* Region 상태 */
  REGN_HEAD_NODE *regionPtr;
                              /* region 헤드 포인터 */
  SIG_MSG_NODE *pRecvedSigList;
                              /* 수신된 signals 리스트 */
  SIG_RCASE_MODE *pSigRecvCase;
                              /* SIGNAL Recv-Case 포인터 */
  unsigned *pSigMsg;   /* 수신된 SIGNAL 메시지 */
  REGISTERS saveArea; /* 레지스터 저장 영역 */
};
```

5.4 프로세스 상태 천이

CHILL 프로세서들이 생성된 후, 각 프로세스는 수행 상황에 따라 (그림 5)와 같은 상태들로 천이 되며 수행된다. 즉, START 실행문에 의해 생성된 CHILL 프로세스는 준비 상태로 Ready 큐에 대기하며, 스케줄러에 의해 실행(Running) 상태로 천이 된다. 실행 상태에서 실행중인 프로세스는EVENT에 대한 DELAY 및 DELAY CASE와 BUFFER에 대한 RECEIVE 및 RECEIVE CASE, 그리고 SIGNAL에 대한 RECEIVE CASE를 위한 실행 조건이 맞지 않으면 지연 상태가 된다. 지연 상태의 프로세스는 다른 프로세스가 EVENT에 대한 CONTINUE, 그리고 BUFFER 및 SIGNAL에 대한 SEND 실행문을 실행하여 실행 조건이 충족되면 다시 준비 상태로 천이 된다. 또한 실행 상태의 프로세스가 STOP 실행문을 실행하면 종료 상태로 천이 되며 실행을 종료하도록 하였다.

특히, 프로세스가 다수의 프로세스에 의해 공유되는 REGION내에 머무는 시간을 최소로 하기 위하여 REGION과 관련된 상태는 별도로 관리되도록 설계하였으며, REGION내에서 대기 상태로 천이 된 프로세스에 대해서는 다른 프로세스보다 우선적으로 처리되도록 구현하였다.



(그림 5) CHILL 프로세스의 상태 천이도

5.5 스케줄링

스케줄링은 다음에 실행될 프로세스를 준비(ready) 큐로부터 선택하는 것이다. CHILL 언어의 병행처리 기능들은 모두 우선 순위에 의한 처리를 기본으로 채택하고 있기 때문에, CRS에서도 우선 순위를 고려한 라운드로빈(round-robin) 스케줄링 정책을 사용하고 우선 순위가 낮은 프로세스의 스케줄링을 보완하기 위하여 추가적으로 난수를 이용한 방법을 사용하도록 설계하였다. 난수를 이용한 스케줄링 방법은 동일한 우선 순위를 가진 프로세스에 대하여 스케줄링 횟수가 현저히 작은 경우 스케줄링의 필요 시점마다 현재 프로세스 개수로 정규화 된 난수를 계산하고 계산된 난수와 동일한 프로세스 인스턴스를 갖는 프로세스를 선택하는 방법이다.

6. 시험 및 결과분석

6.1 커널 및 인터페이스부 시험

구현된 CRS 시스템은 UNIX를 운영체계로 사용하는 VAX-11 계열의 시스템 상에서 커널 부분 시험, 인터페이스 시험, 그리고 응용 프로그램의 실행 시험 등 3단계로 시험하였다.

CRS의 커널부는 CRS의 구동을 통한 CHILL 프로그램의 구동과 문맥 교환을 위한 문맥 저장 및 복구 기능에 대한 시험으로 CRS 구동을 위한 호스트 시스

템과의 인터페이스는 (그림 5)의 CRS 구동 루틴과 C 언어로 작성된 프로그램을 링크하여 해당 C 프로그램이 정상적으로 구동 및 실행되는가를 시험하였다. 또한 호스트 시스템에서 프로그램의 실행 상태를 표현하는 각종 레지스터 값들을 저장하고 복원하는 문맥 교환 기능의 시험을 위하여 (그림 6)의 루틴과 [부록 1]의 문맥 교환 시험 프로그램을 작성하여 각 프로세스의 문맥을 출력시켜 프로세스간 문맥 교환이 정상적으로 실행되는가를 확인하였다.

```
_saveContext(saveArea)
unsigned *saveArea;
{
  asm("movl 4(ap), r0");      / *인수 목록으로부터 저장 영
                                역 획득 */
  asm("movq r2, (r0)+");      /* r2, r3 저장  */
  asm("movq r4, (r0)+");      /* r4, r5 저장 */
  asm("movq r6, (r0)+");      /* r6, r7 저장 */
  asm("movq r8, (r0)+");      /* r8, r9 저장 */
  asm("movq r10, (r0)+");     /* r10, r11 저장 */
  asm("movq 8(fp),(r0)+");    /* 이전의 ap,fp 저장*/
  asm("addl3 $8, ap, (r0)+");/* 이전의 sp 저장 */
  asm("movl 16(fp), (r0)" ); /* 이전의 pc 저장 */
  asm("clrl r0" );            /* r0 클리어 */
}

_resumeContext(saveArea, returnValue)
unsigned *saveArea;
int returnValue;
{
  asm("movl 8(ap), r0");      /* 반환값 */
  asm("movl 4(ap), r1");      /* 인수 목록으로부터 저장 영
                                 역 획득 */
  asm("movq (r1)+, r2");      /* r2, r3 재저장 */
  asm("movq (r1)+, r4");      /* r4, r5 재저장  */
  asm("movq (r1)+, r6");      /* r6, r7 재저장  */
  asm("movq (r1)+, r8");      /* r8, r9 재저장  */
  asm("movq (r1)+, r10");     /* r10, r11 재저장 */
  asm("movq (r1)+, r12");     /* ap, fp 재저장  */
  asm("movl (r1)+, sp");      /* sp 재저장 */
  asm("jmp *(r1) ");          /* pc 재저장과 그곳으로
                                 jump */
}
```

(그림 6) 문맥 저장 및 복원 루틴(Vax-11 용)

시험 결과 CRS에 의해 생성된 각 프로세스에 대한 문맥 교환이 정상적으로 실행됨을 확인하였으며, 실행 결과는 [부록 1]의 시험 결과와 같다.

CHILL 컴파일러와의 인터페이스 시험은 본 논문에서 설계하여 <표 2>에 제시한 CHILL 병행처리 프리미티브 변환 규칙에 따라 구현된 ETRI CHILL 컴파일러의 코드 생성기가 생성한 어셈블리 코드를 확인하였으며, 최종적으로 응용 프로그램 실행 시험을 통하여 그 결과를 확인하였다.

6.2 결과 분석

본 논문에서 구현된 CRS는 설계 목표를 만족하며, 연구목표로 설정한 CHILL의 병행처리 기능 구현에 대한 결과를 확인하였다.

구현된 CRS는 기계 종속도가 매우 낮아 이식성이 높고 각 기능별로 모듈화 되어 있어 수정 및 변경이 용이하며, 병행처리 프리미티브의 추가 및 삭제가 용이한 구조로 설계되었기 때문에 유사한 분야의 병행처리를 위한 실행시간 지원 시스템 구현시 구현 모델로 활용이 가능하다.

7. 결 론

본 논문에서는 ITU-T가 권고한 CHILL 언어에 정의된 병행처리 기능들을 사용하여 작성된 병행 프로그램을 호스트 시스템에서 수행시킬 수 있도록 하기 위한 CRS의 설계 및 구현에 대하여 기술하였다. 제안된 CRS는 CHILL 프로세스간의 동기화와 통신 및 상호배제 등 병행처리를 위한 기능들을 지원하여 VAX-11 계열 및 SUN-3/4 시스템에서 병행 CHILL 프로그램의 실행환경을 제공한다.

CRS는 이식성 및 유지보수성을 높이기 위하여 모듈화 개념으로 설계하였으며 기계 종속적인 부분을 최소화하였다. CRS의 기계 종속적인 부분은 CRS 구동루틴과 문맥교환 루틴으로 전체 프로그램의 약 0.1%미만이며 어셈블리 언어로 구현하였고, 나머지 부분들은 C 언어로 구현하였다. 또한 호스트 시스템 종속성을 최소화하기 위하여 메모리 할당 요구(calloc) 및 메모리 반환(free)과 난수(rand) 함수를 제외하고는 호스트 시스템에서 제공하는 라이브러리 함수를 사용하지 않고 직접 C언어로 구현하였다.

CRS는 라이브러리 형태로 설계 구현되었기 때문에 병행 CHILL 언어로 작성된 응용 프로그램과 CRS와의 인터페이스는 CHILL 병행처리 기능들에 해당하는 CRS의 해당 프리미티브에 대한 프로시듀어 호출 형태로 이루어지며, CHILL 컴파일러의 코드 생성시에 자동적으로 변환될 수 있다.

본 연구를 요약하면 다음과 같다.

1) CRS는 유지보수성 및 확장성을 높이기 위하여 모듈화 개념으로 설계하였으며, 이식성을 높이기 위하여 기계 종속적인 부분을 최소화하였다.

2) 해당 시스템의 어셈블리 언어로 작성된 기계 종속적인 부분들을 독립적인 모듈로 설계하여 다른 기종으로의 전환이 용이하도록 하였다.
3) 구현된 라이브러리 기법을 활용하면 타 프로그래밍 언어의 병행처리 기능 구현용 프로토타이핑 모델로 이용할 수 있다.

앞으로 타이머 관련 처리 기능과 스케줄링 정책을 보완하고 CHILL 언어에 정의된 예외 처리기능을 추가하면 보다 유용한 시스템으로 향상시킬 수 있을 것이다.

[부록 1] 프로세스 문맥교환 시험 프로그램(일부)

```
/* CHILL Run-time System */
main()
{
    int i;
    int exist;
    int outermost();

    for(i = 1; i < MAX_NUM_PROCESSES; i++) {
         pcb[i].pid = -1;
    }

    /* initialization of the loutermost process */
    start(outermost, DFLT_STACKSIZE);
    /* save KERNEL status then resume here */
    _saveContext(&pcb[KERNEL].saveReg);
    /* select next process to be run */
    schedule();
    printf("SELECTED : Process %d WITH\n", this());
    printStatus(this());
    /* resume process */
    _resumeContext(&pcb[this()].saveReg, 1);
}

block()
{
    /* save status of CHILL process */
    if (!_saveContext(&pcb[this()].saveReg)){
      printf("BLOCKED : Process %d\n", this());
      /* go to KERNEL to schedule */
    _resumeContext(&pcb[KERNEL].saveReg, 1);
    }
/* CHILL process blocked will be resumed here */
    printf("RESUMED : Process %d\n", this());
}

schedule()
{
    ......

    while ( 1 ) {
         printf("\nNext process to be run (0-%d) ? ",
              noOfProcesses);
......

start(process, stackSize)
int (*process)();
int stackSize;    /* size in byte */
{
    char *malloc();

    noOfProcesses++; /* assumed pid = 0 for kernel */
    pcb[noOfProcesses].pid = noOfProcesses;
    if (stackSize < DFLT_STACKSIZE)
        stackSize = DFLT_STACKSIZE;
    stackSize
        =(stackSize+sizeof(long)-1)/
            sizeof(long)*sizeof(long);
    pcb[noOfProcesses].saveReg.sp
        =(long)(calloc(1, stackSize) + stackSize);
    ......

/* C 언어에 의해 기술된 CHILL 프로세스들 */
outermost()     /* CHILL main - outermost 프로세스 */
{
    int p2(), p3();
    printf("Executing outermost process firstly...\n");
    start(p2, 1024);
      printf("START... p2 with stack size 1024\n");
    start(p2, 1024);
      printf("START... p3 with stack size 1024\n");
    start(p2, 1024);
      printf("START... p2 with stack size 1024\n");
    start(p2, 1024);
      printf("START... p2 with stack size 1024\n");
    start(p3, 1024);
      printf("START... p3 with stack size 1024\n");
    start(p3, 1024);
      printf("START... p3 with stack size 1024\n");
    for (;;){
      printf("Executing outermost process\n");
      block();
    }
}

p2()                        /* CHILL 프로세스 P2 */
{
    for (;;){
      printf("Executing process definition p2\n");
      block();
    }
}

p3()     /* CHILL 프로세스 P3 */
{
    for (;;){
      printf("Executing process definition p3\n");
      block();
    }
}
```

[실행결과]

```
[CRS@tdx] crs

Next process to be run (0-1) ? 1
SELECTED : Process 1 WITH
STATUS of Process 1 : pc 802 sp 18436 fp 18436
Executing outermost process firstly...
       START... p2 with stack size 1024
       START... p3 with stack size 1024
       START... p2 with stack size 1024
       START... p2 with stack size 1024
       START... p3 with stack size 1024
```

```
      START... p3 with stack size 1024
Executing outermost process
BLOCKED : Process 1

Next process to be run (0-7) ? 2
SELECTED : Process 2 WITH
STATUS of Process 2 : pc 1042 sp 59396 fp 59396
Executing process definition p2
BLOCKED : Process 2

Next process to be run (0-7) ? 3
SELECTED : Process 3 WITH
STATUS of Process 3 : pc 1042 sp 67588 fp 67588
Executing process definition p2
BLOCKED : Process 3

Next process to be run (0-7) ? 4
SELECTED : Process 4 WITH
STATUS of Process 4 : pc 1042 sp 75780 fp 75780
Executing process definition p2
BLOCKED : Process 4

Next process to be run (0-7) ? 2
SELECTED : Process 2 WITH
STATUS of Process 2 : pc 254 sp 59372 fp 59372
RESUMED : Process 2
Executing process definition p2
BLOCKED : Process 2


Next process to be run (0-7) ? ^C[CRS@tdx]
[CRS@tdx]
[CRS@tdx] exit
```

참 고 문 헌

[1] C.M,Chase, A.L.Cheung and M.R.Smith, "Paragon : A Parallel Programming Environment for Scientific Application Application Using Communication Structure," Journal of Parallel & Distributed Computing, Vol.16, No.2, pp.79-91, Oct. 1992.

[2] J.R.Allen and K.Kennedy, "Automatic Translation of Fortran Programs to Vector Form," ACM Transaction on Programming Language and System, pp.491-542, Oct. 1987.

[3] P.B.Hansen. "The Programming Language Concurrent Pascal," IEEE Transaction on Software Engineering, Vol.SE-1, No.2, pp.199-207, Jun. 1975.

[4] H.H.Gehani and W.D.Boome, "Concurrent C," Software-Practice and Experience, Vol.16, No.9, pp.821-844, Sep. 1986.

[5] N.Wirth, "Programming in Modula-2," Springer-Verlag, 1982.

[6] CCITT Recommendation Z.200, "CCITT Hight Level Language(CHILL)," Red Book, CCITT, 1985.

[7] J.Galletly, "OCCAM 2," Pitman, 1990.

[8] US Department of Defense, "Programming Language Ada : Reference Manual," Vol.106, Lecture Notes in Computer Science, Springer-Verlag, 1981.

[9] G.R.Andrew and F.B. Schneider, "Concept and Notations for Concurrent Programming," ACM Computing Surveys, Vol.15, pp.3-43 No.1, 1983.

[10] J.L.Keedy, "The Concurrent Processing Features of The CCITT Language CHILL," A.T.R. Vol.17, No.1, pp.33-51, 1983.

[11] Minoru Kuboto and Norio Sato, "Concurrent CHILL Operating System," Proceedings of ICC'84, Amsterdam, Netherlands, pp.531-534, May 1984.

[12] J.Lenzser and G. Ruckle, "A CHILL-Operating System on Top of VAX/VMS," Proceedings of the 4th CHILL Conference, pp.115-120, Oct. 1986.

[13] A.Camici, V.Giarratana and F.Manucci, "A CHILL Software Development System for Distributed Architectures," Proceedings of ICC'81, Denver, pp.53.2.1-53.2.5, 1981.

[14] A.Camici, et al., "CHILL for Supporting Software Engineering Environments," Proceedings of the 2nd International Conference on Software Engineering for Telecommunication Switching Systems, Jul. 1983.

[15] D.B.Loveman, "High Perofrmance Fortran," IEEE Parallel & Distributed Technology," Vol.1, No.1, pp.25-42, Feb. 1993.

[16] C.M.Pancake and D.Bergmark, "Do Parallel Language Respond to the Needs of Scientific Programmers ?," IEEE Computer, pp.13-23, 1990.

[17] 이준경 외 4인, "A Development of Object-Oriented CHILL Compiling Environment for Telecommunicational Switing Systems", JCCI'96 논문집, pp. 286-290, 1996.

[18] 김상국 외 5인, "A Design of Compiling Environment for CHILL and Object-Oriented CHILL,"

KICS'96 하계 논문집, pp.1210-1213, 1996.
[19] 김명호, 이준경, 이동길, "ECT : CHILL96 언어를 위한 컬렉션 라이브러리", 정보과학회논문지, 제4권 제2호, pp.291-303, 1998.

하 수 철

e-mail : soocha@dragon.tejon.ac.kr
1981년 홍익대학교 컴퓨터공학과 졸업(학사)
1986년 홍익대학교 대학원 컴퓨터 공학과(석사)
1990년 홍익대학교 대학원 컴퓨터 공학과(박사)
1999년~현재 한국정보처리학회 논문지 편집위원
1998년~현재 한국정보처리학회 멀티미디어시스템연구회 부위원장
1997년~현재 소프트웨어연구센터(SOREC) 운영위원
1996년 한국전자통신연구원 초빙연구원
1991년~1992년 플로리다 주립대학교/텍사스 주립대학교 객원교수
1981년~1984년 Army Logistics Command, System Analyst
1987년~현재 대전대학교 컴퓨터공학과 교수
관심분야 : 소프트웨어공학(객체지향화), 멀티미디어 응용, 게임공학, 시각언어, 프로토콜기술언어

조 철 희

e-mail : chcho@etri.re.kr
1982년 홍익대학교 전자계산학과 졸업(학사)
1997년 대전대학교 대학원 컴퓨터 공학과(석사)
1982년 한국전자통신연구원 입소
1982년~현재 한국전자통신연구원(책임연구원)
관심분야 : 소프트웨어 엔지니어링, 객체지향 방법론, 이동통신서비스, Mobile Computing, 망관리(TMN)