5장. 프로세스간 통신 메커니즘 (Interprocess Communication Mechanism)

프로세스들은 상호간의 활동을 조정하기 위해서 프로세스간, 그리고 커널과 통신을 한다. 리눅스는 여러 종류의 프로세스간 통신 기능(Inter-Process Commuication, IPC)을 제공한다. 리눅스는 시그널과 파이프 이외에도 시스템 V IPC를 제공하는데 시스템 V IPC는 이 기능이 처음으로 등장한 유닉스 버전의 이름을 따서 지어진 이름이다.

5.1 시그널(Signal)

시그널은 유닉스 시스템에서 프로세스간 통신을 하는 가장 오래된 방법 중의 하나이다. 이 들은 하나 이상의 프로세스들에게 비동기적인 이벤트를 알리기 위해 사용된다. 시그널은 키 보드 인터럽트로부터 발생되기도 하고, 프로세스가 존재하지 않는 가상 메모리 영역을 사용 하려 하는 경우같은 에러 상황에서도 발생한다. 시그널은 쉘이 자식 프로세스에게 작업 관 리 명령을 보낼 때에도 사용된다.

커널이나 해당하는 권한을 가지고 있는 시스템의 다른 프로세스들이 발생할 수 있는 일련의 정의된 시그널들이 있다. 이러한 시그널들을 보려면 kill 명령을 사용하면 되는데(kill -l), 필 자의 인텔 리눅스 기계에는 다음과 같은 시그널들이 있다.

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGIOT       7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGFEP
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR
```

시그널의 개수는 알파 AXP 리눅스 시스템과 다를 수 있다. 프로세스들은 대부분의 시그널 들을 무시하려면 무시할 수 있지만, 여기에는 두 개의 중요한 예외가 있다: 프로세스의 실 행을 중단시키는 SIGSTOP 시그널과 프로세스를 끝내게 하는 SIGKILL 시그널은 무시할 수 없다. 그렇긴 하지만, 프로세스는 여러가지의 시그널을 어떻게 처리할 지 결정할 수 있다. 프로세스는 시그널을 블럭할 수 있고, 블럭하지 않는 경우에는 스스로 처리하거나 커널이 처리하도록 하는 것 중에 선택할 수 있다. 만약 커널에게 처리를 맡기는 경우에는 시그널에 해당하는 기본 동작이 취해지게 된다. 예를 들어서, 프로세스가 SIGFPE(부동 소수점 연산 예외) 시그널을 받은 경우의 기본 동작은 코어 덤프(core dump)를 하고 프로세스를 끝내는 것으로 되어 있다. 시그널에는 본래 우선순위가 없다. 한 프로세스에게 동시에 두 개의 시그 널이 발생하는 경우, 이 시그널들이 프로세스에 전달되는 순서나 처리되는 순서는 정해

져 있지 않다. 또한 동시에 같은 시그널이 여러번 발생하는 것을 처리할 수 있는 메커니즘도 없다. 따라서, 프로세스가 SINGCONT 시그널을 한번을 받든 42번을 받든 이를 구별할 방법 이 없다.

리눅스는 프로세스의 task_struct에 저장된 정보를 사용해서 시그널 기능을 구현한다. 지 원할 수 있는 시그널의 갯수는 프로세서의 워드(word) 크기에 제한을 받는다. 32비트 워드를 사용하는 시스템에서는 최대한 32개의 시그널을 지원할 수 있고, 알파 AXP와 같이 64비트 프로세서를 사용하는 경우에는 최대 64개의 시그널을 지원할 수 있다. 현재 처리 대기중인 시그널들은 signal 항목에 저장되며, 블럭된 시그널들의 마스크는 blocked 항목에 담기 게 된다. SIGSTOP과 SIGKILL을 제외한 다른 모든 시그널들은 블럭킹 할 수 있다. 블럭된 시그널이 발생할 경우 그 시그널은 블럭킹을 해제할 때까지 대기 상태로 남아 있게 된다. 리눅스는 또한 발생할 수 있는 모든 시그널들을 프로세스가 어떻게 처리하는가에 대한 정보 를 가지고 있는데, 이 정보는 프로세스의 task_struct에 있는 sigaction 자료구조의 배열에 저장된다. sigaction에는 여러가지 다른 정보들과 함께, 시그널 핸들러의 주소, 또는 프로세스가 해당 시그널을 무시할 것인지 혹은 커널이 그 시그널을 대신 처리하게 할 것인 지를 나타내는 플래그가 들어 있다. 프로세스는 시스템 콜을 통해서 기본 시그널 핸들러를 바꿀 수 있으며, 이 시스템 콜은 해당 시그널의 sigaction과 blocked 마스크를 변경한 다.

시스템 내의 프로세스들이 모두 다른 프로세스로 시그널을 보낼 수 있는 것은 아니다. 커널 과 관리자는 모든 프로세스에게 보낼 수 있지만, 일반 프로세스는 같은 uid와 gid를 갖는 프 로세스, 또는 같은 프로세스 그룹[1] 내의 프로세스에게만 시그널을 보낼 수 있다. 시그널은 task_struct내 signal 항목의 해당하는 비트를 설정하여 발생된다. 프로세스가 그 시그 널을 블럭하지 않았고, 인터럽트 가능한 상태에서(즉 INTERRUPTIBLE 상태에서) 대기중에 있다면, 프로세스는 현재 상태를 실행중(RUNNING)으로 바꾸고 자신을 실행큐에 넣음으로 써 깨어나게 된다. 이런 방법으로 시스템이 다음번 스케쥴링을 수행할때, 스케쥴러가 그 프 로세스를 실행할 후보로 생각하게 된다. 기본 동작으로의 시그널 처리만이 필요하다면 리눅 스는 시그널 처리을 최적화 시킬 수 있다. 예를 들어 SIGWINCH(X 윈도우가 포커스를 변경) 가 발생하였고 기본 핸들러를 사용할 것이라면, 프로세스가 따로 수행할 일은 없게 되는 것 이다.

시그널은 발생하는 순간 바로 프로세스로 전달되는 것이 아니라 그 프로세스가 다시 수행될 때까지 기다려야 한다. 즉 프로세스가 시스템 콜을 마치고 돌아올 때마다 signal과 blocked가 매번 검사되는데, 이때 블럭되지 않은 시그널이 존재하는 경우 비로서 프로세스 로 전달되는 것이다. 이 방식은 상당히 신뢰성이 낮은 방법처럼 보이지만, 시스템 내의 프로 세스들은 무슨 목적에서든(예를 들면 터미널에 문자를 찍기 위해서) 실행 시간 대부분에 걸 쳐 시스템 콜을 계속 수행하므로 그렇지는 않다. 원한다면 프로세스는 시그널 발생을 기다 리는 것을 선택할 수 있는데, 이 경우 인터럽트 허용 상태에서 시그널이 전달되어 올 때까 지 프로세스는 멈춰 서있게 된다. 리눅스 시그널 처리 코드는 현재 블럭되지 않은 시그널에 대해서 sigaction 자료구조를 참조한다.

시그널 핸들러가 기본 핸들러로 되어 있으면 커널이 그 처리를 대신 수행하게 된다. SIGSTOP 시그널에 대한 기본 핸들러는 현재 프로세스의 상태를 중지됨(STOPPED)으로 바 꾸고, 새로 실행할 프로세스를 선택하기 위해 스케쥴러를 실행한다. SIGFPE 시그널을 받으 면 커널은 현재 프로세스를 코어 덤프하고 프로세스를 종료한다. 이와 달리 프로세스가 직 접 자신의 시그널 핸들러를 지정했을 수도 있다. 이것은 시그널이 발생할 때마다 호출되는 것으로, sigaction 자료구조가 이 루틴의 주소를 가지고 있다. 이제 커널은 반드시 프로세 스의 시그널 핸들러를 호출해야 하는데, 이것이 어떻게 이루어지는가는 프로세서에 따라 다 르지만, 한가지 사실, 즉 현재 프로세스는 커널 모드에서 실행중이며 곧 사용자 모드에서 커 널 혹은 시스템 루틴을 부른 프로세스로 돌아가려고 한다는 점만은 모든 CPU들이 염두에 두고 대처하여야 하는 문제이다[2]. 이 문제는 프로세스의 스택과 레지스터를 조작함으로써 해결가능하다. 프로세스의 프로그램 카운터를 그 시그널 처리 루틴으로 설정하고, 핸들러로 전달할 인

자를 스택 프레임에 추가하거나 레지스터에 담아 보내는 것이다. 이후 프로세스가 실행을 재개하면 시그널 처리 루틴은 마치 정상적인 방법으로 호출되었던 것같이 보이게 된 다.

리눅스는 POSIX 호환이므로, 프로세스는 특정 시그널 처리 루틴이 호출되었을 때 어떤 시 그널을 블럭할 것인지를 지정할 있다. 이것은 프로세스 시그널 핸들러가 불리는 동안 blocked 마스크의 값을 바꾸게 됨을 뜻한다. blocked 마스크는 시그널 처리 루틴이 종료 될 때 원래 값으로 돌려 놓아야 한다. 그래서 리눅스는 정리용 루틴을 하나 더 불러서, 시그 널을 받은 프로세스의 콜 스택에 저장해놓은 원래의 blocked 마스크 값을 꺼내어 복구하 도록 한다. 또한 여러 시그널 처리 루틴이 계속 호출되어야 할 필요가 있을 때는 이 루틴들 을 스택처럼 쌓아서, 한 핸들러를 빠져나오면 다음 핸들러가 호출되고, 마지막으로 정리용 루틴이 호출되도록 시그널 처리를 최적화한다.

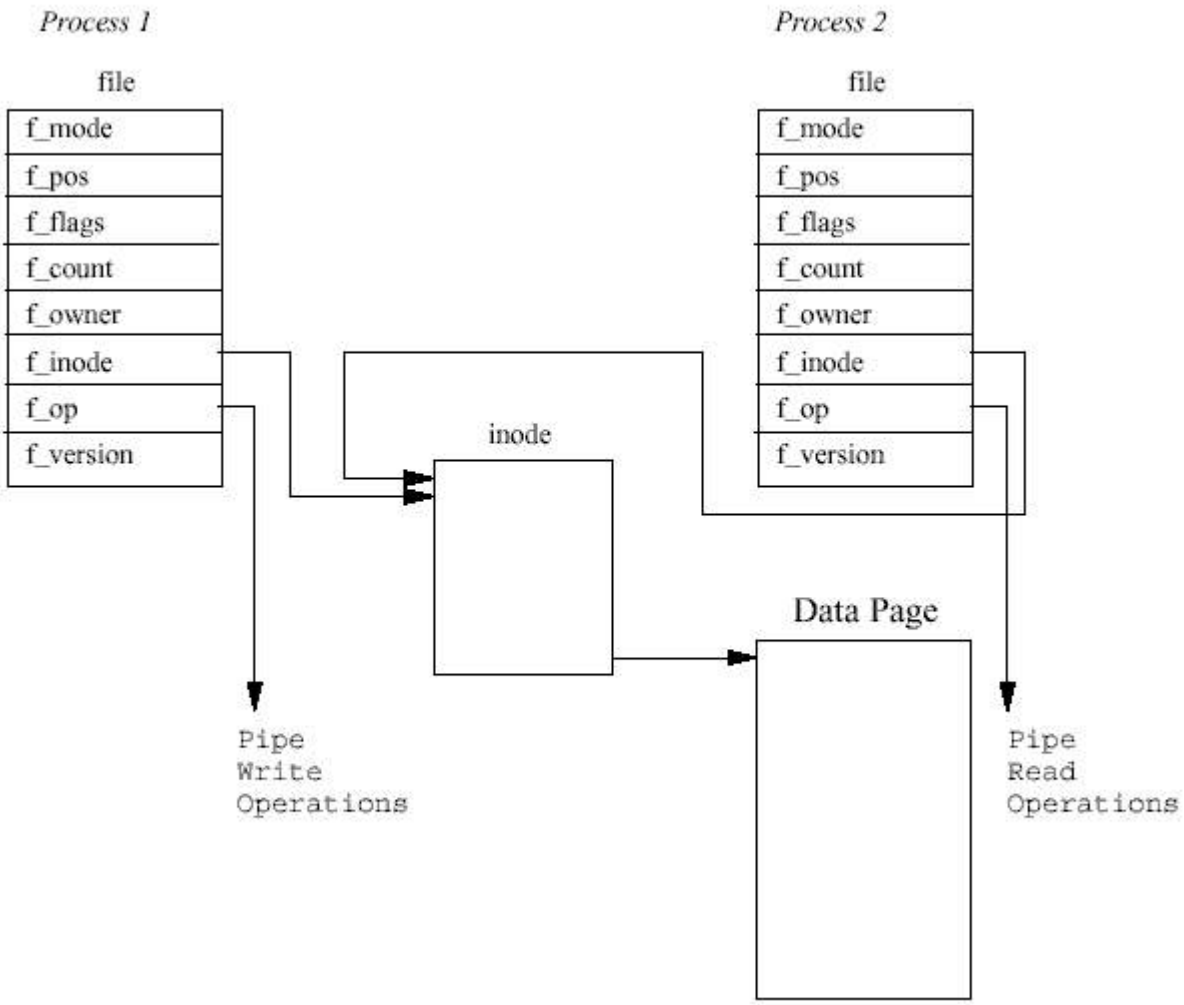

그림 5.1 : 파이프

5.2 파이프(Pipe)

일반적으로 사용하는 리눅스 쉘들은 모두 리다이렉션(redirection)을 지원한다. 예를 들어

```
$ ls | pr | lpr
```

이라는 명령은 ls 명령이 출력하는 파일 이름들을 pr 명령의 표준 입력으로 보내고, pr 명령 은 입력된 내용을 페이지 단위로 나눈다. pr 명령의 표준 출력으로 나온 결과는 다시 lpr 명 령의 표준 입력으로 보내져서 기본 프린터로 출력된다. 파이프는 위의 예에서처럼 한 프로 세스의 표준 출력을 다른 프로세스의 표준 입력으로 보내주는 단방향 바이트 스트림이다. 파이프로 연결되는 프로세스들은 이런 리다이렉션이 일어나고 있다는 것은 알지 못하며, 보 통 때와 마찬가지로 동작한다. 여기서 프로세스간에 임시 파이프를 만들어 연결시켜주는 것 은 쉘이다.

리눅스에서 파이프는 임시로 만들어진 VFS inode를 똑같이 가리키는 두 개의 file 자료구 조를 사용해서 구현되며, 여기서 VFS inode는 메모리상의 물리적 페이지를 가리키게 된다[3]. 그림 5.1은 각 file 자료구조가 각기 다른 파일 연산 루틴 벡터를 가리키는 포인터를 가지 고 있는 모습을 보여준다. 여기서 한 file 자료구조는 파이프에 쓰는 함수에 대한 포인터 를, 다른 자료구조는 파이프에서 읽어들이는 함수에 대한 포인터를 가진다. 이것은 보통의 파일에 읽고 쓰는 시스템 콜이 아래 계층의 차이에 관계없이 동작하도록 한다[4]. 쓰는 프로 세스가 파이프에 쓴 데이터는 공유 데이터 페이지에 복사되고, 읽는 프로세스가 그 파이프 로부터 읽어 들일때는 공유 데이터 페이지로부터 데이터가 복사되게 된다. 리눅스는 파이프 에 대한 접근을 동기화해야 한다. 파이프의 읽는 프로세스와 쓰는 프로세스가 반드시 차례 를 지킬 수 있도록 해야 하고, 그렇게 하기 위해 락(lock)과 대기큐(waiting queue), 시그널 등 을 사용한다.

프로세스가 파이프에 쓰기를 할 때는 쓰기를 하는 표준 라이브러리 함수를 사용한다. 이들 함수들에는 파일 기술자(file descriptor)를 넘기는데, 이는 프로세스가 가진 여러개의 file 자 료구조(이들 각각은 프로세스가 열어 놓은 파일을 나타내며, 이 경우에는 열어 놓은 파이프 를 나타낸다)에 대한 인덱스이다[5]. 리눅스 시스템 콜은 이 파이프를 나타내는 file 자료구 조에서 가리키고 있는 쓰기 루틴을 사용한다. 이 쓰기 루틴은 쓰기 요청을 처리하기 위해 파이프를 나타내는 VFS inode에 있는 정보들을 이용한다. 파이프에 요청한 바이트들을 모두 쓸 공간이 있고, 파이프를 읽는 프로세스가 락을 걸어두지 않았다면, 리눅스는 먼저 파이프 에 락을 걸고, 쓸 데이터 바이트들을 프로세스의 주소공간에서 공유 데이터 페이지로 복사 한다. 만약 읽는 프로세스가 파이프에 락을 걸어두었거나 데이터를 담을 충분한 공간이 없 다면, 현재 프로세스는 해당 파이프 inode에 있는 대기큐에 들어가 잠들고, 실행할 수 있는 다른 프로세스를 선택하기 위해 스케쥴러를 호출한다. 잠든 프로세스는 인터럽트 허용 상태 이므로, 시그널을 받을 수 있으며, 읽는 프로세스에 의해 쓸 데이터를 담기에 충분한 공간이 생기거나 파이프의 락을 풀리면 깨어나게 된다. 데이터를 쓰고 나면 파이프의 VFS inode의 락을 풀고, inode의 대기큐에서 기다리며 잠들어 있는 읽는 프로세스를 깨우게 된다.

파이프에서 데이터를 읽는 과정은 파이프에 쓰는 과정과 매우 비슷하다. 프로세스들은 블럭 킹을 하지 않고 읽을 수 있는데 (이는 파일이나 파이프를 열 때 어떤 모드를 사용하였느냐 에 따라 다르다[6]), 이 경우 읽을 데이터가 없거나 파이프에 락이 걸려있으면 에러가 돌아온 다. 이는 프로세스가 잠들지 않고 실행을 계속할 수 있다는 것이다. 블럭킹 모드라면 파이프 inode의 대기큐에서 쓰기 프로세스가 끝나기를 기다려야 한다. 양쪽 프로세스가 파이프를 통 한 작업을 종료하면, 파이프 inode는 공유 데이터 페이지와 함께 폐기된다.

리눅스는 지정 파이프(named pipe)도 지원한다. 지정 파이프는 FIFO라고도 불리는데 이는 파 이프가 먼저 들어온 것이 먼저 나가는(First In First Out, FIFO) 원칙에 따라 동작하기 때문이 다. 파이프에 먼저 쓴 데이터는 파이프에서 읽을 때 먼저 나온다. 파이프와 달리 FIFO는 임 시적으로 생성된 것이 아니라 파일 시스템에 실재 존재하는 것이며, mkfifo 명령으로 생성 할 수 있다. 프로세스는 해당하는 접근 권한을 가지고 있다면 FIFO를 자유롭게 사용할 수 있다. FIFO를 여는 방법은 파이프와는 조금 다르다. 파이프(두개의 file 자료구조와 이들이 가진 VFS inode, 공유 데이터 페이지)는 한번에 만들어지는데 반해, FIFO는 이미 존재하는 것이며, 사용자에 의해 열고 닫혀지는 것이다[7]. 리눅스는 FIFO에 쓰는

프로세스가 없을 때 다른 프로세스가 이를 읽기 위해 열려고 하는 것이나, FIFO에 쓰는 프로세스가 FIFO에 쓰기 를 하기 전에 읽는 프로세스가 읽으려고 하는 것 모두 처리해야 한다. 이를 제외하면, FIFO 는 거의 완전히 파이프와 똑같은 방법으로 취급되며, 같은 자료구조와 연산을 사용한다[8].

5.3 소켓(Socket)[9]

REVIEW NOTE : 네트워크 장을 쓴 다음에 추가한다.

5.3.1. 시스템 V IPC 메커니즘

리눅스는 유닉스 System V (1983)에서 처음 등장한 세가지 종류의 프로세스간 통신 방법을 제공한다. 이들은 메시지 큐(message queue)와 세마포어(semaphore), 그리고 공유 메모리 (shared memory)이다. 이들 시스템 V IPC 방법들은 모두 똑같은 인증 방법을 공유한다. 프로 세스는 커널에 시스템 콜로 이들 자원을 가리키는 유일한 참조 식별자(reference identifier)를 전달함으로써만 이들에 접근할 수 있다. 이들 시스템 V IPC 객체들에 대한 접근은 접근 권 한(access permission)을 가지고 검사하는데, 파일에 대한 접근을 검사하는 것과 많이 비슷하 다. 시스템 V IPC 객체에 대한 접근 권한은 시스템 콜을 통하여 객체의 생성자에 의해 지정 된다. 각 통신 방법들은 참조 식별자를 자원 테이블에 대한 인덱스처럼 사용하는데, 참조 식 별자는 말그대로 인덱스인 것은 아니고, 인덱스를 만들기 위해서는 약간의 계산이 필요하다.

시스템에 있는 시스템 V IPC 객체를 나타내는 리눅스 자료구조는 모두, 프로세스의 소유자 와 생성자의 uid, gid와 이 객체에 대한 접근 모드(소유자, 그룹, 그밖에 대한)와 IPC 객체의 키를 가진 ipc_perm 이라는 자료구조를 포함하고 있다. 키는 시스템 V IPC 객체의 참조 식 별자를 찾는 한 방법으로 쓰인다. 모두 두 종류의 키를 지원하는데, 공용(public)와 개인용 (private)이 그것이다. 만약 키가 공용라면 시스템에 있는 어떤 프로세스든지 권한 검사를 통 과한다면 시스템 V IPC 객체에 대한 참조 식별자를 찾을 수 있다[10]. 시스템 V IPC 객체는 키 로 참조할 수 없으며, 이들에 대한 참조 식별자로만 참조할 수 있다.

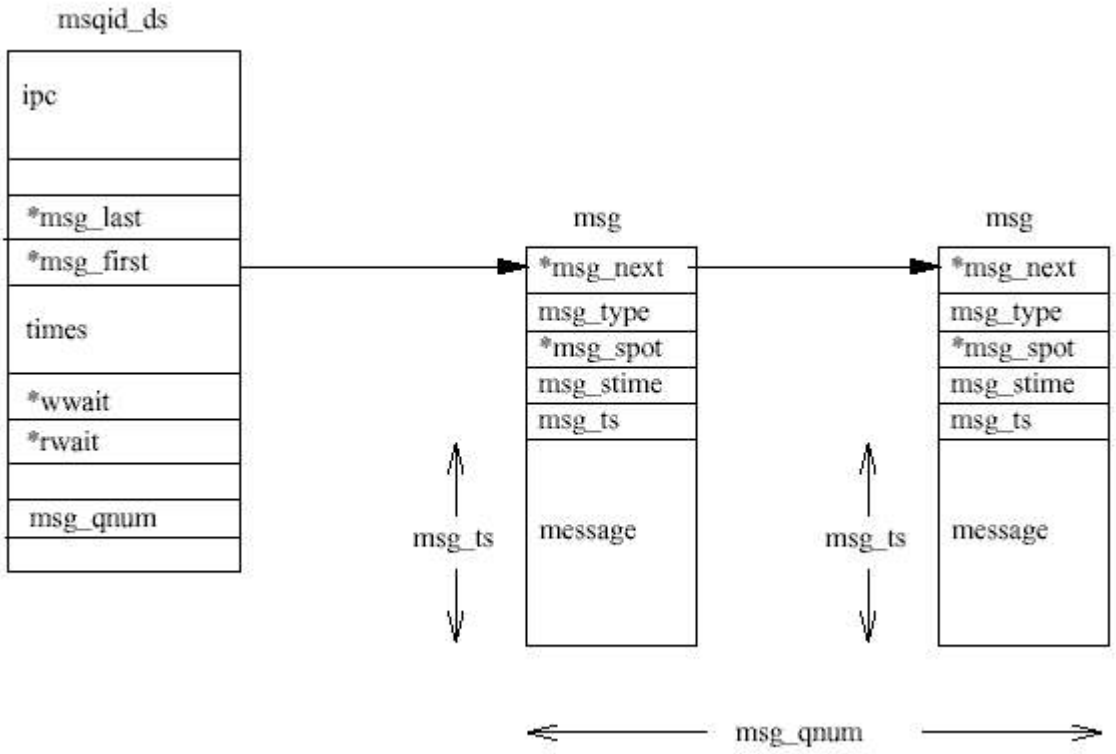

그림 5.2 : System V IPC 메시지 큐

5.3.2 메시지 큐(Message Queue)

메시지 큐는 하나 이상의 프로세스가 메시지를 쓸 수 있고, 이를 하나 이상의 프로세스가 읽을 수 있을 수 있도록 한다. 리눅스는 메시지 큐의 리스트를 msgque 벡터로 관리한다. msgque의 각 원소는 메시지 큐에 대한 모든 것을 기술하는 msqid_ds 자료구조를 가리킨 다. 메시지 큐를 하나 생성하면 msqid_ds 자료구조를 시스템 메모리에서 할당받아 이 벡 터에 삽입한다.

각 msqid_ds 자료구조는 ipc_perm 자료구조와, 이 큐에 들어온 메시지에 대한 포인터들 을 가지고 있다. 추가로, 리눅스는 큐에 마지막으로 쓴 시간같은 큐 수정 시간도 유지한다. msqid_ds는 두 개의 대기큐도 가지고 있다 : 하나는 큐에 쓰려는 프로세스를 위해, 하나는 큐에서 읽을려는 프로세스를 위해서다.

프로세스가 큐에 메시지를 쓰려고 할 때마다, 효력 사용자 식별자(effective user identifier)와 효력 그룹 식별자(effective group identifier)를 큐의 ipc_perm 자료구조에 있는 모드와 비교한 다. 그래서 프로세스가 큐에 쓸 수 있다면 메시지는 프로세스의 주소공간에서 msg 자료구 조로 복사되고 메시지 큐의 마지막에 놓인다. 각 메시지에는 같이 협동하는 프로세스간에 서로 약속한 타입인, 응용프로그램 지정 타입을 꼬리표로 단다. 리눅스는 쓸 수 있는 메시지 의 개수와 길이를 제한하고 있으므로 메시지를 쓸 공간이 없을 수도 있다. 이런 경우 프로 세스는 메시지 큐의 쓰기 대기큐(msqid_ds의 *wwait 항목)에 추가되고 실행할 새로운 프 로세스를 선택하기 위해 스케쥴러를 호출한다. 프로세스는 메시지 큐에서 하나 이상의 메시 지가 읽혔을 때 깨어나게 된다.

큐에서 읽는 것은 비슷한 과정을 거친다. 마찬가지로 프로세스가 가진 큐에 대한 접근 권한 을 검사한다. 읽는 프로세스는 타입에 관계없이 큐에 있는 첫번째 메시지를 가져올 지, 또는 특정한 타입을 가진 메시지를 선택할 지 고를 수 있다. 이 기준에 맞는 메시지가 없다면 읽 으려는 프로세스는 메시지

큐의 읽기 대기큐(msgq_id의 *rwait 항목)에 추가되고, 스케쥴 러가 실행된다. 큐에 새로운 메시지를 쓰게 되면 이 프로세스는 깨어나 다시 실행할 수 있 게 된다.

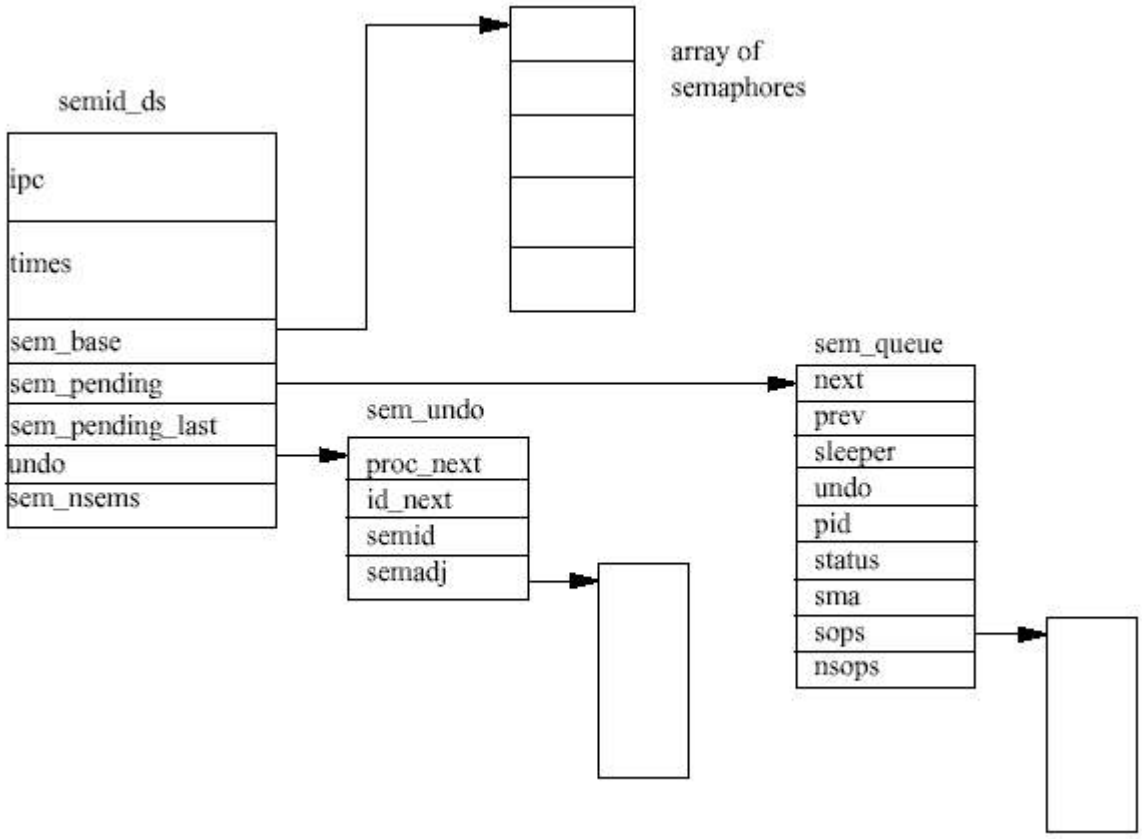

그림 5.3 : System V IPC 세마포어

5.3.3 세마포어(Semaphore)

세마포어의 가장 단순한 형태는 메모리의 한 위치에 있는 변수로, 그 값을 하나 이상의 프 로세스가 검사하고 설정(test and set)할 수 있는 것이다. 이 검사 및 설정(test and set) 연산은, 각 프로세스에 있어서, 중단될 수 없는, 즉 원자성을 가진 것이다. 즉 한번 시작되면 아무것 도 이를 중단할 수 없다[11]. 이 검사 및 설정 연산의 결과는 세마포어의 현재값에 더하여 값 을 설정하는 것이며, 이 값은 양수일 수도 음수일 수도 있다. 검사 및 설정 연산의 결과에 따라서 한 프로세스는 다른 프로세스가 세마포어의 값을 바꿀 때까지 기다리며 잠들어야 할 수도 있다. 세마포어는 동시에 한 프로세스만이 실행해야 하는 중요한 코드가 있는, 임계지 역(critical region)을 구현하는데 사용할 수 있다.

여러개의 협동하는 프로세스가 하나의 데이터 파일에서 레코드를 읽거나 쓴다고 하자. 이 때 파일에 대한 접근이 완전히 조화롭게 이루어지길 바랄 것이다. 여기서 세마포어를 사용 할 수 있는데, 먼저 세마포어의 초기값을 1로 하고, 파일 연산을 하는 코드의 주위에 두개의 세마포어 연산을 두어서, 첫번째 것은 세마포어의 값을 검사하고 값을 감소시키고, 다음 것 은 값을 검사하고 증가시키게 할 수 있다. 파일에 접근하려는 첫번째 프로세스는 세마포어 의 값을 감소시키려고 하고, 이것이 성공하여 세마포어의 값은 0이 된다. 이 프로세스는 이 제 계속 진행하여 데이터 파일을 사용하지만, 이를 사용하려고 하는 다른 프로세스는 세마 포어의 값을 감소시키려고 했는데 결과가 -1이 되므로 실패한다. 이 프로세스는 첫번째 프 로세스가 데이터 파일 작업을 끝마칠 때가지 중단될 것이다. 첫번째 프로세

스가 데이터 파 일 작업을 마치면 세파포어의 값을 다시 증가시켜 1로 만든다. 이제 기다리는 프로세스는 깨어나서 이번에는 세마포어를 감소시키려는 시도가 성공하게 된다[12].

시스템 V IPC 세마포어 객체들은 각각 세마포어의 배열을 나타내고, 리눅스는 이를 나타내 기 위해 semid_ds 자료구조를 사용한다. semary는 시스템에 있는 모든 semid_ds 자료구 조를 가리키고 있는, 포인터의 벡터이다. semid_ds 자료구조에는 sem_nsems 갯수만큼의 세마포어 배열이 있으며, 각각은 sem 자료구조로 기술된다. 이 세마포어 배열은 sem_base 이 가리키고 있다[13]. 시스템 V IPC 세마포어 객체의 세마포어 배열을 관리할 수 있는 권한을 가진 모든 프로세스들은 이들을 다루는 시스템 콜을 부를 수 있다. 시스템 콜은 한번에 여 러개의 연산을 지정할 수 있으며, 각 연산은 세가지 입력 - 세마포어 인덱스, 연산 값, 플래 그들의 세트 - 으로 나타내진다[14]. 세마포어 인덱스는 세마포어 배열에서의 인덱스이며, 연 산 값은 세마포어의 현재 값에 추가될 숫자 값이다. 먼저 리눅스는 모든 연산이 성공할 수 있는지 테스트한다. 연산 값을 세마포어의 현재 값에 더한 값이 0 이상이거나, 연산 값과 세마포어의 현재 값이 모두 0일 때, 이 연산은 성공하게 된다. 만약 세마포어 연산의 하나라도 실패한다면 리눅스는 프로세스를 중단할 수 있는데, 이는 시스템 콜을 부를 때 플래그에 블 럭킹 모드를 사용하지 않을거라고 지정하지 않은 경우이다. 프로세스가 중단되어야 한다면 리눅스는 수행해야 할 세마포어 연산의 상태를 저장하고, 현재 프로세스를 대기큐에 넣는다. 이 작업은 sem_queue 자료구조를 스택에 만들어 이것의 내용을 채움으로써 이루어진다[15]. 새 sem_queue 자료구조는 세마포어 객체의 대기 큐의 끝에 놓여진다 (여기서 sem_pending과 sem_pending_last 포인터를 사용한다). 현재 프로세스는 sem_queue 자료구조에 있는 대기큐(sleeper 항목)에 놓여지고, 실행할 다른 프로세스를 고르기 위해 스케쥴러가 호출된다.

만약 모든 세마포어 연산이 성공하여 프로세스가 중단될 필요가 없다면, 리눅스는 계속 진 행하여 세마포어 배열의 올바른 멤버에게 연산을 적용한다. 리눅스는 이제 대기큐에서 기다 리며 중단되어 있는 프로세스들이 이 세마포어 연산에 적용될 수 있는지 검사해야 한다. 리 눅스는 연산 미결큐(sem_pending)의 각 멤버를 차례로 살펴보고, 이번엔 세마포어 연산이 성공할 수 있는지 알아보기 위한 테스트를 한다. 만약 성공한다면 연산 미결 리스트에서 sem_queue 자료구조를 제거하고 세마포어 배열에 그 세마포어 연산을 적용한다. 리눅스는 잠든 프로세스를 깨워 다음번 스케쥴러가 실행될 때에는 다시 시작할 수 있도록 만든다. 리 눅스는 미결 리스트를 처음부터 시작하여 더이상 세마포어 연산을 적용할 수 없고, 깨울 프 로세스가 없을 때까지 계속 살펴본다.

세마포어에는 한가지 문제가 있는데 데드락(deadlock)이 바로 그것이다. 이는 한 프로세스가 임계지역에 들어가면서 세마포어의 값을 바꾸었는데 프로세스가 잘못되거나 강제로 종료되 어서 이 임계지역을 빠져나가지 못한 경우에 발생한다[16]. 리눅스는 이런 문제를 세마포어 배 열에 대한 조정 리스트를 관리함으로써 막는다. 이 개념은 이런 조정을 적용하면 세마포어 가 그 프로세스가 세마포어 연산을 수행하기 이전의 상태로 되돌아가게 하는 것이다. 조정 에 대한 것은 sem_undo 자료구조에 보관되고, 이들은 semid_ds 자료구조와 세마포어 배 열을 사용하는 프로세스의 task_struct 양쪽에 큐된다.

각 개별적인 세마포어 연산은 조정을 관리하도록 요구할 수 있다. 리눅스는 프로세스마다 각 세마포어 배열에 대해 많아봐야 하나의 sem_undo 자료구조를 관리한다. 만약 연산을 요청한 프로세스가 이 자료구조를 가지고 있지 않다면 필요할 때 하나 생성할 것이다. 새로 만들어진 sem_undo 자료구조는 이 프로세스의 task_struct 자료구조와 세마포어 배열의 semid_ds 자료구조 양쪽에 큐된다. 세마포어 배열에 있는 세마포어에 연산을 적용하면 연 산값을 반대로 한 값이 이 프로세스의 sem_undo 자료구조에 있는 조정 배열의 세마포어 엔트리로 추가된다. 즉 연산값이 2를 더하는 것이었다면 이 세마포어의 조정 엔트리에는 -2 가 더해진다.

프로세스가 종료하여 지워질 때, 리눅스는 sem_undo 자료구조 세트를 가지고 세마포어 배 열에 조정을 적용한다. 만약 한 세마포어 세트가 지워지면 프로세스의 task_struct의 큐 되어 있는 sem_undo 자료구조는 그대로 남아있지만, 세마포어 배열 식별자는 잘못된 것일 것이다. 이 경우 세마포어 정리 코드는 간단하게 sem_undo 자료구조를 무시한다.

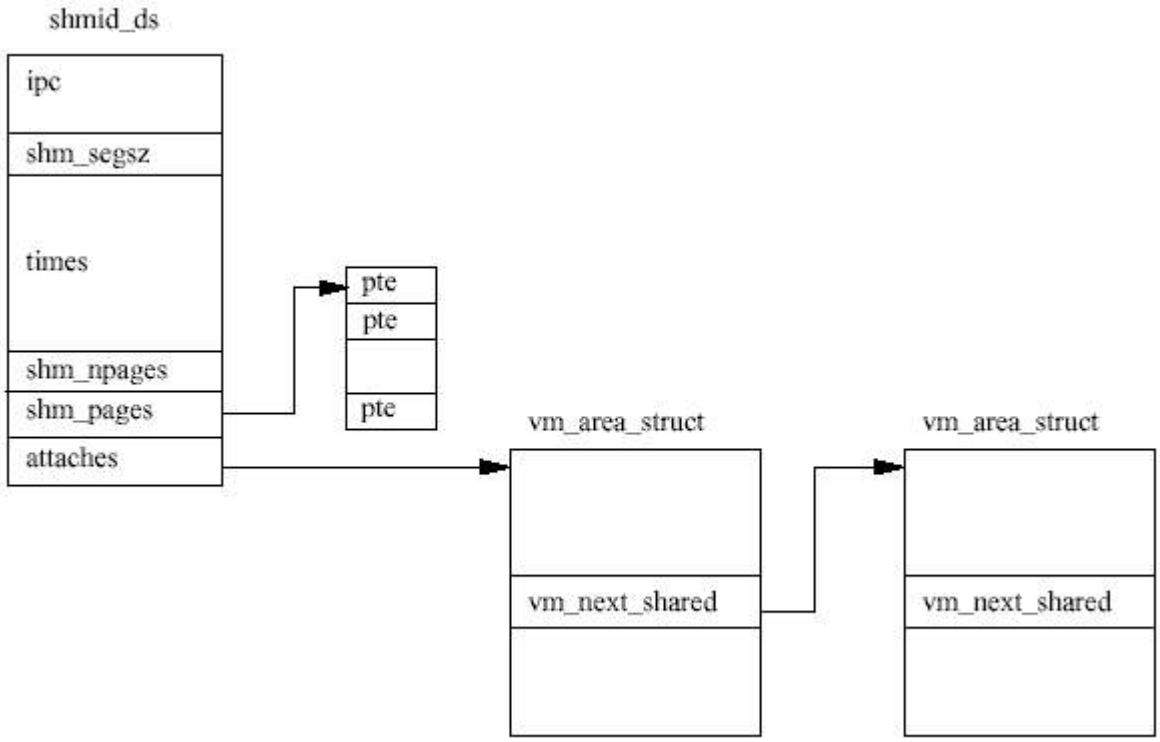

그림 5.4 : System V IPC 공유 메모리

5.3.4. 공유 메모리(Shared Memory)

공유 메모리는 하나 이상의 프로세스들이 자신들의 가상 주소 공간에 공통으로 나타나는 메 모리를 통하여 통신할 수 있도록 한다. 이들 프로세스의 페이지 테이블 각각에는 이 공유 가상 메모리 페이지들을 가리키는 페이지 테이블 엔트리가 있게 된다. 이들은 모든 프로세 스의 가상 메모리에서 똑같은 주소에 있을 필요는 없다. 다른 시스템 V IPC 객체와 마찬가 지로 공유 메모리 영역로의 접근은 키에 의해 제어되고 접근 권한을 검사하게 된다. 하지만 한번 메모리가 공유되고 나면 프로세스들이 이를 어떻게 사용하는지에 대해서 아무런 검사 도 하지 않는다. 프로세스들은 다른 방법, 예를 들어 시스템 V 세마포어같은 것을 사용하여 메모리로의 접근을 동기화하여야 한다.

새로 만들어진 공유 메모리 영역은 shmid_ds 자료구조로 나타낸다. 이들은 shm_segs 벡 터에 저장된다. shmid_ds 자료구조는 공유 메모리 영역이 얼마나 큰지, 얼마나 많은 프로 세스가 사용하고 있으며, 공유 메모리가 프로세스의 주소공간에 어떻게 매핑되어 있는지에 대한 정보를 가진다. 공유 메모리를 만든 프로세스가 이 메모리에 대한 접근권한과 키가 공 용인지 개인용인지 제어하며, 충분한 권한만 있다면 공유 메모리를 물리적인 메모리로 락[17] 시킬수도 있다.

메모리를 공유하길 바라는 각 프로세스들은 시스템 콜을 통하여 이 가상 메모리에 연결해야 한다. 이것은 이 프로세스에서의 공유 메모리를 기술하는 새로운 vm_area_struct 자료구 조를 만들어낸다[18].

프로세스는 공유 메모리가 자신의 가상 주소 공간에 위치할 곳을 선택할 수도 있고, 아니면 리눅스가 충분히 큰 빈 영역을 선택하도록 할 수도 있다. 새로 만들어진 vm_area_struct 자료구조는 shmid_ds 가 가리키고 있는 vm_area_struct 리스트에 추 가된다. vm_newxt_shared와 vm_prev_shared 포인터들은 이들을 서로 연결하는데 사용 한다. 가상 메모리는 이렇게 연결하는 동안에 실제로 만들어지지 않으며, 처음으로 프로세스 가 여기에 접근하려고 할 때 만들어진다.

프로세스가 공유하고 있는 가상 메모리의 한 페이지에 처음으로 접근을 시도하면 페이지 폴 트가 발생한다. 리눅스가 이 페이지 폴트를 처리할 때 이를 기술하는 vm_area_struct 자 료구조를 발견하게 된다. 여기에는 이 타입의 공유 가상 메모리에 대한 처리 루틴에 대한 포인터가 있다[19]. 공유 메모리의 페이지 폴트 처리 코드는 shmid_ds의 페이지 테이블 엔트 리를 뒤져서, 공유 가상 메모리의 해당 페이지에 대한 페이지 테이블 엔트리가 있는지 찾 는다. 만약 없다면 물리적 메모리를 하나 할당 받아 이를 나타내는 페이지 테이블 엔트리 를 만들 것이다. 이를 현재 프로세스의 페이지 테이블에 넣으면서 shmid_ds에도 저장한다. 그래서 다음 프로세스가 이 메모리에 접근하려고 하다가 페이지 폴트가 발생하면, 공유 메 모리 페이지 폴트 처리 코드가 이를 찾아서, 새로 만들어진 물리적인 페이지를 그 프로세스 에게도 사용하게 한다. 따라서 공유 메모리의 어떤 페이지에 접근하는 첫번째 프로세스는 이를 생성하고, 다른 프로세스들이 여기에 접근할 때는 이를 자신의 가상 메모리 공간에 추 가하게 된다.

프로세스가 더이상 가상 메모리를 공유하길 바라지 않을 때는 여기로의 연결을 끊는다. 이 메모리를 사용하는 다른 프로세스가 존재하는 한은 연결을 끊는 것은 단지 해당 프로세스에 게만 영향을 미친다. 그 메모리의 vm_area_struct는 shmid_ds 자료구조에서 제거되고 해제될 것이며, 프로세스가 공유하는데 사용했던 가상 메모리 영역을 무효한 것으로 나타내 기 위해 프로세스의 페이지 테이블이 갱신된다. 마지막으로 메모리를 공유하고 있던 프로세 스가 연결을 끊으면 물리적인 메모리에 존재하고 있는 모든 공유 메모리 페이지들은 해제되 고, 이 공유 메모리를 나타내던 shmid_ds 자료구조도 해제된다.

공유 가상 메모리가 물리적인 메모리로 락되어 있지 않을 때 약간 복잡한 문제가 발생한다. 이는 메모리의 사용량이 많아서 공유 메모리가 스왑 디스크로 스왑된 것인 경우도 있다. 공 유 메모리가 어떻게 물리적인 메모리에서 스왑되어 나가거나 들어오는지는 3장에서 설명하 고 있다.

번역 : 이승, 이호, 김진석, 김기용, 심마로
정리 : 이호

---

1) REVIEW NOTE : 프로세스 그룹을 설명할 것

역주 2) 이러한 문제는 시그널이 발생했는지 확인하는 것이 커널 모드에서이고, 시그널 핸 들러는 사용자 모드에서 실행되어야 하기 때문에, 커널 모드에서 바로 시그널 핸들러는 부를 수 없어서 발생한다. 이에 사용자 모드로 바꾸어 시그널 핸들러를 부르고 이것이 끝났을 때 마치 시스템 콜을 한 것처럼 커널 모드로 다시 돌아오게 하는 방법을 쓰는 것 이다. (flyduck)

역주 3) 즉 파이프를 위해 파일 시스템에 파일을 생성하는 것이 아니라, 메모리 상에 공유 메모리 페이지를 만들어서 VFS inode가 이를 사용하게 하는 것이며, 이는 파이프의 속도 를 빠르게 한다. (flyduck)

역주 4) VFS inode의 f_op 항목은 읽기, 쓰기를 포함하여 함수에 관련된 연산들의 포인터 배열이며, 파일 연산은 이 항목에 있는 함수 포인터를 부름으로써 이루어진다. 여기서 파 이프를 읽어들이는 파일을 타나내는 file 자료구조는 보통 파일에 읽는 함수에 대한 포 인터가 아니라 파이프에서 읽어들이는 포인터를 가리키게 하더라도, 시스템 콜에 있어서 는 어짜피 f_op 항목의 함수 포인터를 이용하므로 아무런 차이가 없는 것이다. 이것은 파이프에 쓰는 file 자료구조에 있어서도 마찬가지며, 이러한 기법은 리눅스 커널 곳곳 에서 쓰이고 있다. (flyduck)

역주 5) task_struct는 파일에 관련된 자료구조인 struct files_struct files 항목 을 가지고 있으며, files_struct는 struct file *fd[NR_OPEN] 항목을 가지고 있다. 이 fd에서의 인덱스가 파일 기술자이며, fd 항목은 프로세스에서 열은 파일에 대한 정보 를 가지고 있다. (flyduck)

역주 6) file 자료구조의 f_flags 항목이 파일에 관련된 플래그를 가지고 있는데, 여기에 O_NONBLOCK가 지정되어 있으면 블럭킹을 하지 않는 상태이다. 이는 파일을 열때 지정 할 수도, ioctl이나 fcntl같은 시스템 콜을 통해서 바꿀 수도 있다. (flyduck)

역주 7) 앞의 ls | pr을 지정 파이프를 사용한다면, 우선 mkfifo fifo (파일이름은 다르게 지 정해도 된다) 명령으로 지정 파이프를 만든 후, ls > fifo & pr < fifo로 하면 된다. (flyduck)

역주 8) FIFO도 파이프와 마찬가지로 실제로 파일을 통하여 통신하는 것이 아니라 메모리에 공유 페이지를 만든다. 즉 읽고 쓰는 것은 파이프와 똑같은 방법을 사용하게 된다. FIFO 와 파이프의 차이점은 파이프는 임시로 생성되는 것인데 반해 FIFO는 이미 만들어져 있 는 것이므로 두 개의 프로세스가 open(), close() 함수를 통하여 파일처럼 열 수 있으므 로 쉘이 관여하지 않아도 통신할 수 있다는 점이다. (flyduck)

역주 9) 소켓은 네트워킹에서 이야기하는 소켓이다. 소켓은 한 컴퓨터 내에서 프로세스 사 이에 통신을 할 수 있게 할 뿐만 아니라 네트웍에 있는 다른 컴퓨터에 있는 프로세스와 도 통신을 가능하게 한다. 소켓의 사용은 유닉스에서 전통적으로 사용하는 파일 기술자 (file descriptor)를 통하여 한다. 즉 socket() 함수를 부르면 소켓을 나타내는 파일 기술자 가 돌아오고 이를 가지고 bind, listen, connect, accept 등의 연산을 할 수 있으 며, 파일과 마찬가지로 read, write, close 연산을 할 수 있다. (flyduck)

역주 10) 공용키가 아니라 개인용키를 사용하는 경우는 ipc_perm의 key 항목이 IPC_PRIVATE로 지정되는데, 이를 가지고 참조 식별자를 찾을 수 없다. 개인용키를 사용 하는 IPC 개체는 개체 번호를 통해서만 접근할 수 있다. (flyduck)

역주 11) 검사를 하고 설정하는 사이에 중단이 되어 다른 것이 실행되었는데 여기서 이 값 을 검사하고 설정한다면, 다시 이전으로 돌아와서 설정하려고 할 때는 이미 값이 바뀐 이후가 되어 문제가 발생할 것이다. 따라서 이 검사 및 설정 연산은 중단되어서는 안되 며, CPU에서 제공하는 특별한 명령어를 이용하거나 운영체제 코드를 통하여 구현된다. 리눅스에서는 커널모드에서 비선점형이므로 이 연산이 중단되지 않는다고 생각하고 일반 연산으로 처리한다. 커널 코드에서 세마포어 값을 검사하는 함수는 try_semop()이며, 세 마포어 값을 바꾸는 함수는 do_semop()이다. 이들은 ipc/sem.c에 있다. (flyduck)

역주 12) 첫번째 프로세스가 세마포어의 값을 0으로 바꾼 후에는 다음 프로세스가 검사단계 에서 실패하므로 세마포어의 값은 변하지 않으며, 첫번째 프로세스가 이 값을 1로 바꾼 후에야 기다리고 있던 프로세스가 이를 바꿀 수 있게 된다. 그래서 세마포어의 값은 항 상 0보다 크거나 같게 된다. (flyduck)

역주 13) 하나의 세마포어 객체에 여러개의 세마포어 배열(sem의 배열)이 있으며, 시스템 콜은 이 중의 일부만을 검사하고 설정할 수 있다. 임계지역에서 여러개의 세마포어를 필 요로하는 경우 이 중에 필요로 하는 것들만 지정할 수 있다. (flyduck)

역주 14) 세마포어 연산을 하는 시스템 콜은 sys_semop(int semid, struct sembuf *sops, unsigned nsops)이며, 여기서 하나의 연산을 가리키는 sembuf는 sem_num, sem_op, sem_flg 세가지 원소로 이루어져 있다. 이 세마포어 연산은 세마포어 값을 감 소시킬 수도, 증가시킬 수도 있다. (flyduck)

역주 15) 메시지 큐의 경우는 대기큐가 task_struct를 가지고 있는 단순한 wait_queue 의 연결 리스트로 되어 있지만, 세마포어의 대기큐는 sem_queue의 연결 리스트로 되어 있다. 이는 메시지 큐의 경우 읽기를 기다리는지, 쓰기를 기다리는지만 구별하면 되지만, 세마포어에서는 세마포어 연산으로 넘겨준 인자들을 모두 저장하고 있어야 하고 좀 더 복잡한 연산이 필요하기 때문에 이 정보를 모두 sem_queue에 저장하는 것이다.(flyduck)

역주 16) 데드락은 이 경우뿐만 아니라 한 프로세스가 필요로 하는 자원을 다른 프로세스가 사용하고 있어 대기 상태로 갔는데, 나중에 그 프로세스가 앞의 프로세스가 점유하고 있 는 자원을 필요로 하게 되어 프로세스들이 서로 상대가 자원 사용을 종료하기만을 기다 리게 되는 상태도 포함한다. 이것은 단순히 두 프로세스가 아니라 여러 프로세스가 꼬리 에 꼬리를 물고 있을 때 복잡하게 이루어질 수 있다. 단순히 세마포어에 국한하여 이야 기한다면, 프로세스가 세마포어를 이용하여 임계지역으로 들어간 후 다시 세마포어를 얻 으려고 하지 않는다면 이런 문제는 발생하지 않을 것이다. (flyduck)

역주 17) 가상 메모리를 물리적인 메모리에 존재하게 만드는 것을 말한다. (flyduck)

역주 18) 앞의 프로세스 장에서 설명한 것과 같이 한 프로세스가 할당받은 메모리들은 vm_area_struct의 리스트와 AVL 트리로 관리되는데, 이는 페이지 폴트가 발생했을 때 해당 페이지가 실제 프로세스가 사용하는 메모리인지, 어떻게 물리적인 페이지를 만들것 인지 알기 위해 사용된다. (flyduck)

역주 19) nopage 연산 (flyduck)