사용자:KimMinKwan/연습장

목차

과제 1-1 getppid

Design

getppid()라는 system call 함수를 만들어, 함수 호출 시 프로세스의 pid를 출력하도록 구현했습니다. getppid() system call 함수를 구현하기 위해서 xv-6 내부에 있는 함수 중, proc.h, 와 proc.c 파일과 기존에 존재하는 getpid() system call 함수를 참고하여 구현을 했습니다.

Implement

기존의 system call 함수인 getpid()의 내용은 myproc()이라는 함수의 pid 를 return하여 현재 프로세스의 pid를 출력합니다. 이를 참고하여 proc.c 에 있는 system call 함수 중 myproc()을 분석한 결과, 이는 구조체 type의 함수로서 proc.h 에 선언된 struct proc 에 맞춰 프로세스의 정보를 반환하는 함수 입니다.

proc 구조체 내부를 분석 하면서 프로세스 관련 여러가지 정보 중, int pid 와 struct proc * parent 가 있는 것을 확인하고, myproc()을 호출하여 현재 프로세스의 정보를 담은 구조체 정보를 얻은 다음, 그 안의 parent 의 pid 정보를 받아온다면, 프로세스의 부모 pid를 가져올 수 있다는 것을 확인했습니다. 그리하여, getppid() system call 함수는 myproc()->parent->pid 로 부모의 pid를 반환하고, 이에 따른 wrapper 함수인 sys_getppid()에서는 getppid()를 반환하도록 하여 system call 함수를 만들었습니다.

Result

user program 으로 porject01 이라는 파일을 만들고 Makefile에 project01을 추가하여 xv-6 터미널에 project01을 입력했을 때, project01이라는 user program이 실행 되도록 먼저 설정을 했습니다. 그 후, project01안에 과제의 목적인 프로세스의 pid와 부모 프로세스의 pid를 출력하도록 system call 함수인 getpid() 함수와 새로만든 getppid() 함수를 호출하여 각각 int pid 와 int ppid 에 저장한 후, 이를 print 하도록 하였습니다. make를 완료한 후, xv-6 터미널에서 project01을 입력했을 때, 정상적으로 프로세스의 pid와 부모 pid가 모두 출력 되는것을 확인했습니다.

Trouble Shooting

getppid() 과제를 진행하면서 발생한 문제점으로는 헤더 파일이었습니다. 새로 getppid.c 라는 파일을 만든 후, myproc() 함수와 struct proc 을 가져오기 위해서는 proc.h의 헤더파일을 포함하는 것이 필요했습니다. 하지만, 그냥 proc.h 만을 했을 때는 redefinition 이나 다른 헤더파일이 추가로 필요하다는 에러가 매번 나와 추가로 필요한 헤더파일을 xv-6 내부 파일에서 찾아가면서 추가했습니다. 이 문제점을 해결하면서 xv-6 내부에 있는 다른 system call 함수나 선언되어있는 헤더 파일을 찾아보고 알아보게 되었습니다.

과제 1-2 xv-6

과제 1-1인 getppid() system call을 구현 하면서, xv-6 내부에 있는 파일의 코드를 많이 알아보게 되었습니다.

우선 xv-6 란 연습용 운영체제 프로그램으로서, 이를 사용할시, 새로운 system call 함수를 구현하거나 기존에 있는 system call 함수에 관련하여 찾아볼 수 있습니다.

저는 그 중에서, proc.h 와 proc.c 에 관련하여 많은 것을 알게 되었습니다. 이 두 파일은 프로세스에 관련된 system call 함수가 구현된 곳으로, 여기에는 fokr() 새로운 프로세스를 만든 후 이의 pid를 반환하는 함수, exit() 좀비 상태에 있는 현재 프로세스를 종료하는 함수, wait() 자식 프로세스가 종료되기를 기다리고 그것의 pid를 반환하는 함수 등 여러가지를 공부하고 알게 되었습니다.

그 중, proc.h 에 선언된 proc 구조체 관련하여 공부하게 됬습니다. 이 구조체는 전역변수로 선언되어 프로세스 관련 system call 함수 모두에 쓰이는 구조체로, 프로세스의 기본 정보를 담고 있습니다. 그 정보로는 프로세스의 메모리 크기, pid, 상태 등이 있으며 그 상태로는 6가지 UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE 가 있습니다. 이를 공부하면서, 듣기만 했던 좀비 상태나 그 외의 상태에 대해 알게되었고, 이 상태에 따라 system call 함수의 역할과 기능도 다양하다는 것을 알게 되었습니다.

xv-6에 새로운 system call 함수를 만들면서 이 과정에 대한 이해 또한 추가로 알게 되었습니다. system call 관련 내용의 함수를 구현하고, 그 함수를 호출하는 wrapper 함수를 만들고, wrapper 함수를 syscall.h 와 syscall.c 에 추가하여 make를 하면 작성한 system call 함수를 사용할 수 있습니다. 더 자세히는 system call을 할 함수 파일의 명을 Makefile에 추가한 다음, defs.h에 새로만든 system call 함수를 추가하고 wrapper 함수를 위에 설명한 것처럼 syscall.h 와 syacall.c에 추가합니다. 그리고, user program을 만들어, 그 파일 또한 Makefile에 추가한 후, make 와 make fs.img 를 입력하여 정상적으로 make를 진행한 후, xv-6 터미널에 user program을 입력하면 새로만든 system call 함수를 사용할 수 있습니다.

이 과정에서 새로알게된 사실로는 getpid 처럼 system call 함수가 반드시 필요하지는 않다는 것입니다. 굳이 system call 함수와 wrapper 함수 두개를 만들 필요 없이 wrapper 함수에 system call 함수의 내용을 넣어 make 하여도 정상적으로 동작한다는 점 입니다.

지금은 프로세스 관련한 system call 함수만 봐왔지만, 내가 흔히 터미널에서 새로운 파일을 만드는 명령어인 mkdir 등 여러가지 함수가 xv-6에 내장되어 있으며, 이를 하나씩 알아가고 공부하도록 노력하겠습니다. 감사합니다.
