**Pintos Project 1: User Program (1)**

담당 교수 : 김영재 교수님

조 / 조원 : 개별 프로젝트 / 20201614 이민석

개발 기간 : 2024.09.12~2024.09.24

1. **개발 목표**

* 초기 pintos에는 system call, system call handler, argument passing, user stack등이 구현되어 있지 않다. 이 기능들은 User Program을 kernel이 load하고 execute할 수 있도록 process를 생성하고 관리하는 데에 필요한 부분이다. 또한 System call을 통해 User Program이 kernel과 interaction할 수 있으며, 이 과정에서 허용된 범위의 메모리에 접근할 수 있도록 해야 한다. 따라서 이번 User program 프로젝트에서 앞서 말한 내용들을 구현하는 것이 전반적인 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Argument Passing
   * + User Program은 충분히 여러 개의 인자를 가질 수 있기에, 이를 적절히 parsing한 후, 80x86 calling convention에 맞춰 메모리에 할당한다. 이 때, ‘hex\_dump()’ 함수를 이용하여 stack에 저장된 상태를 확인할 수 있다.
3. User Memory Access
   * + User Program이 사용하는 메모리는 User Memory 영역이어야 한다. 이 때, virtual memory system을 이용하게 되는데, 해당 process의 (\*esp) pointer가 user virtual memory 영역에만 접근해야 한다. 만약 Unmapped virtual memory 영역에 접근하거나, kernel virtual memory 영역에 접근한다면, 이를 허용되지 않는 메모리 접근이라 판단하고 예외 처리를 통해 프로그램을 종료한다.
4. System Calls
   * + User Program이 실행되기 위해선 kernel의 도움이 필요하며, kernel의 기능을 사용하기 위해 다양한 system call 함수가 필요하다. System call handler를 통해 User Memory Access를 확인한 후, 필요한 인자들을 system call function으로 넘겨주며 system call function을 호출한다. 이번 프로젝트에서 구현한 system call function으론, ‘halt()’, ‘exit()’, ‘exec()’, ‘wait()’, ‘write()’, ‘read()’, ‘fibonacci()’, ‘max\_of\_four\_int()’ 가 있다.
   1. **개발 내용**
5. Argument Passing
   * User Program 실행에 있어서, 전체 command line을 적절히 parsing하고, Kernel에서 stack에 적절히 할당해야 한다. 따라서 Argument passing 과정은 크게 Parsing과 Stack Push로 나눌 수 있다.

Parsing 단계에선, 인자들을 “ “(공백)을 기준으로 잘라 ‘argv[]’에 개별 문자열로 저장한다. 인자 개수를 나타내는 변수’ argc’ 값도 구하게 된다.

Stack push 단계에선, 80x86 calling convention에 맞춰 각 ‘argv[]’들을 스택에 쌓아야 한다. 호출 규약에 따르면, stack에 역순으로 쌓아야 하기 때문에 ‘argv[]’의 역순부터 stack에 push한다. 우선, ‘argv[]’의 문자열을 삽입하고, 4byte align을 맞춰준 후, ‘argv[]’의 문자열이 위치한 주소를 삽입한 후, 인자 개수인 ‘argc’를 삽입하고, 마지막으로 return address를 삽입한다.

1. User Memory Access
   * Pintos는 Memory 영역을 User Memory와 Kernel Memory로 구분한다. 이 때 Physical Memory를 직접적으로 사용한다면, 메모리 관리에 있어서 어려움이 존재하기에 pintos를 비롯한 많은 OS는 Virtual Memory System을 사용하게 된다. 각 process는 각자의 memory 영역을 가지게 되고, VM 또한 user virtual memory와 kernel virtual memory로 나뉘게 된다.

이 때 User program이 physical memory에 mapped되지 않은 Unmapped Virtual memory에 접근하거나, User mode에서 Kernel virtual Memory에 접근하거나, 잘못된 포인터 또는 NULL 포인터를 참조하려는 경우 page fault가 발생한다.

이러한 Invalid memory access를 방지하기 위해, User program의 pointer를 kernel에서 사용하기 전에, Valid한 memory를 access하고 있는지 체크해야 한다. 이는 thread/vaddr.h 헤더파일에 있는 ‘is\_user\_vaddr()’ 함수와 ‘is\_kernel\_vaddr()’ 함수를 이용하여 확인할 수 있다. 또한 thread/palloc.c 에 구현된 ‘palloc\_get\_page()’함수를 통해 해당 주소가 실제로 유효한 virtual memory에 mapping 되어있는지 확인할 수 있다.

1. System Calls
   * 2번의 Invalid Memory Access와 관련하여, pintos와 같은 OS는 user program이 kernel memory 영역에 접근하지 못하도록 한다. 이는 kernel의 file system, memory 관리, process 제어 등의 권한을 user program에게서 방지하려는 목적이다. 그러나 user program의 실행에 있어서 앞서 말한 권한들이 필요할 수 있다. 이를 해결하는 방법이 바로 system call이다. System call functions들을 통해 user program이 파일을 읽고 쓰거나, 프로그램을 실행할 수 있게 된다.
   * 이번 프로젝트에서 개발할 system call function들은 ‘halt()’, ‘exit()’, ‘exec()’, ‘wait()’, ‘write()’, ‘read()’, ‘fibonacci()’, ‘max\_of\_four\_int()’ 총 8가지이다. 다음은 각각의 함수 skeleton과 간략한 설명이다.
     1. void halt (void): ‘devices/shutdown.h’에 선언된 ‘shutdown\_power\_off()’함수를 호출하며 Pintos를 종료한다.
     2. void exit (int status): 현재 실행중인 User program을 종료하고, 종료 상태인 status를 kernel에 return한다. status == 0 일 경우에는 성공을, status != 0 인 경우 오류를 의미한다.
     3. pid\_t exec (const char \*cmd\_line): cmd\_line의 명령어를 실행하고, 해당 process의 pid를 반환한다. 이 때 생성된 process는 자식 process이며, 부모 process는 자식 process가 성공적으로 load 되었는지 확인해야 한다. 이를 위해 적절한 synchronization이 필요하다.
     4. int wait (pid\_t pid): 자식 process가 종료될 때까지 wait하는 과정을 구현해야 한다. 자식 process의 pid가 valid한지 확인한 후, 자식 process의 종료 상태를 return한다.
     5. int read (int fd, void \*buffer, unsigned size): ‘fd’로부터 ‘input\_getc()’함수를 이용해 최대 ‘size’만큼의 byte를 읽어들여, buffer에 저장한다. 읽은 byte 수를 return하며, 이번 프로젝트에선 fd == 0 인 경우만 고려한다.
     6. int write (int fd, const void \*buffer, unsigned size): ‘put\_buf()’함수를 이용해 buffer의 내용을 한 번에 최대 ‘size’만큼의 byte를 fd에 적는다. 기록된 byte 수를 return하며, 이번 프로젝트에선 fd == 1 인 경우만 고려한다.
     7. int fibonacci(int n): n번째 피보나치 수를 계산하여 return한다.
     8. int max\_of\_four\_int(int x, int y, int z, int w): x, y, z, w 중 가장 큰 정수를 return한다
   * 다음은 User level에서 system call API를 호출한 이후 kernel을 거쳐 다시 User level로 돌아올 때까지의 각 요소를 설명하는 내용이다.
   * User Program이 system call API를 호출한다면, kernel의 기능을 사용할 수 있다. 우선 system call number와 함수의 인자들이 stack에 push된다. 여기서 말하는 system call number는 “lib/syscall-nr.h”에 구현된 number를 말한다. 인자 개수에 맞는 syscall이 호출되면, interrupt를 발생시키는 int $0x30이 실행된다. 이를 통해 kernel mode로 전환되며, interrupt handler가 호출된다. “threads/interrupt.c”에 정의된 ‘intr\_handler()’가 실행되면, 내부적으로 system call handler를 호출한다. “userprog/syscall.c”에 구현된 “syscall\_handler()”함수는 넘겨받은 control 권한을 통해 struct intr\_frame의 esp 포인터를 통해 스택에 접근할 수 있게 된다. 80x86 calling convention에 의해 저장된 스택에 따라, 아까 저장했던 system call number와 인자들에 접근이 가능해진다. System call number에 맞게 적절한 함수를 호출하고, 인자를 넘겨주게 된다. System call 처리가 끝나게 되면, 해당 system call function의 결과 값을 struct intr\_frame의 eax에 저장한다. 이를 통해 다시 User level로 돌아오게 된다.
2. **추진 일정 및 개발 방법**
   1. **추진 일정**
   * 09/12 ~ 09/14: Pintos 운영체제의 전반적인 이해와 프로젝트 명세서 및 pintos pdf 내용 숙지. User program이 어떻게 실행되는지에 대한 code level flow 학습 및 Virtual Memory 공부
   * 09/16: Argument Passing 구현 및 User Memory Access에 대한 구현 설계
   * 09/17 ~ 09/19: System call handler 및 system call functions 구현, User Memory Access 코드에 적용.
   * 09/21 ~ 09/26: system call function 중 ‘wait()’과 ‘exec()’ 재구성, Additional Implementation에 관련된 내용 구현 및 테스트 진행.
   1. **개발 방법**
3. Argument Passing
   * “src/userprog/process.c”의 ‘load()’함수에서 Argument를 parsing하고 stack에 저장하는 코드를 구현한다. 현재 실행중인 process에 대해 page directory를 할당한 후, parsing을 거치게 되는데, User program의 실행 인자로 받게 된 file\_name을 적절히 ‘argv’ 배열에 저장한다. 그 후, 해당 실행파일을 open한다.
   * Stack에 인자들을 80x86 calling convention에 맞춰 쌓아야 한다. (\*esp)를 Parsing한 argument 크기만큼 감소하며 인자를 스택에 저장하고, 마지막 stack address의 4byte align을 지켜야 한다. 또한 argv[]가 위치한 address, argc, return address들을 저장한다.
4. User Memory Access
   * User Program이 Valid한 address만 접근할 수 있도록 “src/userprog/syscall.c”에 ‘check\_valid\_addr()’함수를 구현하였다. ‘check\_valid\_addr()’함수는 인자로 넘겨받은 address가 NULL이 아닌지, user memory영역인지, page directory에 mapped 된 address인지 확인한 후, 만약 invalid하다면, exit(-1)을 통해 프로그램을 종료하게 된다. 이 함수를 이용하여 ‘syscall\_handler()’함수에서 system call function들을 호출하기 전, 함수에 인자를 넘기기 전에, 해당 인자들이 valid한 address에 위치하는지 확인한 후, 함수를 호출한다.
   * 새로 구현한 함수는 “src/userprog/syscall.h”에 새로 선언한다.
5. System Calls
   * system call handler 역할을 하는 ‘syscall\_handler()’ 함수는 system call function을 호출하기 위한 handler이며, 이를 위해 User Memory Access를 확인한 후, syscall number에 맞게 해당 함수를 호출한다.
   * 이번 프로젝트에서 구현해야 하는 6가지 system call function인 ‘halt()’, ‘exit()’, ‘exec()’, ‘wait()’, ‘write()’, ‘read()’와 추가적으로 구현하는 ‘fibonacci()’, ‘max\_of\_four\_int()’ 이 2가지 함수를 “/userprog/syscall.c”에 구현한다. 이에 맞게 “lib/syscall-nr.h” 에 맞는 system call number를 새로 선언한다.
   * ‘fibonacci()’, ‘max\_of\_four\_int()’는 이번 프로젝트에서 추가적으로 구현하는 system call function인 만큼, “lib/user/syscall.c”에 인자 개수에 맞는 syscall을 호출하며 새로 선언해야 한다. 이에 맞게 “lib/user/syscall.h”를 수정해야 한다. 또한 ‘max\_of\_four\_int()’는 인자가 4개이므로, “lib/user/syscall.c”에 syscall4를 새로 define해야 한다.
   * System call function 중 ‘exec()’와 ‘wait()’ 은 각각 ‘process\_execute()’, ‘process\_wait()’를 호출하기에 “src/userprog/process.c”에 해당 함수들을 수정해야 한다. ‘process\_execute()’에는 file\_name을 적절히 parsing하여, thread의 name으로 설정해야 하는 요소가 추가되어야 하고, ‘process\_wait()’는 부모 process가 자식 process 종료 전에 terminate되지 않도록 wait하는 부분이 구현되어야 한다. 자식 종료를 위해 대기하는 용도인 semaphore, 또한 자식 메모리를 안전하게 해제할 수 있는 semaphore를 “threads/thread.h”의 thread 구조체에 추가한다. 또한 부모 process가 자식 list를 관리할 수 있도록 필요한 변수들을 “threads/thread.h”의 thread 구조체에 추가한다. 또한 thread가 생성될 시, 해당 변수들이 초기화하는 과정을 “threads/thread.c”의 ‘init\_thread()’함수에 구현한다. 또한 앞서 사용한 semaphore 변수들을 process가 종료되면 적절히 up & down을 하는 코드를 ‘process\_exit()’에 구현해야 한다.
6. **연구 결과**
   1. **Flow Chart**
7. **II. B. 개발 내용에 대한 Flow Chart를 작성**
8. 텍스트, 도표, 평면도이(가) 표시된 사진

   자동 생성된 설명Argument Passing ( Parsing / Stack push 2개로 구분)

Argument Parsing 과정의 flow chart

텍스트, 도표, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Argument를 Stack에 push하는 과정의 flow chart

1. 텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

   자동 생성된 설명User Memory Access

User memory access를 확인하는 ‘check\_valid\_addr()’의 flow chart

1. 도표, 텍스트, 평면도, 스케치이(가) 표시된 사진

   자동 생성된 설명System Calls

System call이 호출된 후 과정을 나타내는 flow chart

* 1. **제작 내용**

1. 텍스트, 스크린샷, 폰트이(가) 표시된 사진

   자동 생성된 설명Argument Passing ( Parsing / Stack push 2개로 구분 )
   * 우선 구현에 필요한 변수들을 선언하고 초기화 한다. ‘file\_ptr’은 ‘file\_name’을 복사해 임시로 저장할 버퍼이다. 복사하는 과정에서 pintos의 기본 라이브러리 함수인 ‘strlcpy()’를 사용하였다. ‘file\_name’을 “ “(공백)단위로 parsing 하여 우선 argument의 개수를 센 후, 이를 argc 값으로 저장한다. 이 과정에서 사용한 ‘strtok\_r()’ 함수는 reentrant한 ‘strtok()’함수이며 이 또한 Pintos 기본 라이브러리에 저장되어 있다. 그 후, ‘malloc()’을 이용하여 동적할당으로 argc의 크기만큼 ‘char \*argv[]’를 저장하려 했으나, pintos 문서의 “malloc() 함수 사용 지양”을 확인하고, 크기를 고정하여 할당하였다. 그 후, 다시 ‘file\_name’이 복사된 ‘file\_ptr’을 tokenizing하여 각 argument들을 argv[]에 저장하였다. 마지막으로 첫번째 argument인 실행 파일 이름을 ‘filesys\_open(argv[0])’으로 넘겨 파일을 열게 된다. ‘filesys\_open()’ 또한 Pintos의 파일 시스템 함수로, 인자로 넘겨받는 파일명을 실행하고, 파일 포인터를 반환한다.
2. int total\_len=0, temp\_len=0;
4. for(i=argc-1;i>=0;i--){
5. temp\_len = strlen(argv[i])+1;
6. (\*esp) -= temp\_len;
7. total\_len += temp\_len;
8. memcpy((\*esp),argv[i],temp\_len);
9. argv[i] = (\*esp);
10. }
12. // word-align
13. for(i=(4-(total\_len%4));i!=0;i--){
14. (\*esp)--;
15. \*(uint8\_t \*)(\*esp) = 0;
16. }
17. // argv[argc] = 0 stack에 push(위와 구분)
18. (\*esp) -= sizeof(char \*);
19. memset((\*esp),0,sizeof(char \*));
20. // argv[argc--] 하면서 argv[i]가 적혀있는 address stack에 push
21. for(i=argc-1;i>=0;i--){
22. (\*esp) -= sizeof(char \*);
23. memcpy((\*esp),&argv[i],sizeof(char \*));
24. }
25. // argv의 시작 addr
26. char \*\*argv\_addr = \*esp;
27. \*esp -= sizeof(char \*\*);
28. \*(char \*\*\*)(\*esp) = argv\_addr;
29. // argc
30. (\*esp) -= sizeof(int);
31. \*(int \*)(\*esp) = argc;
32. // ret addr
33. (\*esp) -= sizeof(void \*);
34. memset((\*esp),0,sizeof(void \*));
35. // hex\_dump((uintptr\_t)(\*esp),(\*esp),128,1);
    * 80x86 calling convention에 따라 argv[argc-1]부터 argv[0]까지 인자들을 역순으로 stack에 저장한다. ‘strlen(argv[i]) + 1’를 계산하고 축적하며 후에 4byte align에 이용한다. esp또한 ‘strlen(argv[i]) + 1’만큼 감소시킨 후, 그 위치에 해당 argv[i]를 저장한다. 복사한 stack address를 다시 argv[i]에 저장하여 인자의 주소를 stack에 저장할 때 쓸 수 있도록 한다.
    * Stack은 항상 4byte align되어 있어야 하므로, ‘total\_len’값을 기준으로 (4 - (total\_len % 4))을 통해 정렬에 남은 byte를 계산할 수 있다. 다음으로 argv[argc] = 0을 통해 NULL을 삽입하여, argument의 끝을 나타낸다.
    * 이전에 argv[i]에 저장했던 argv[i]가 저장된 스택 주소를 다시 스택에 저장한다. 이를 통해 argument의 주소를 참조할 수 있도록 한다. 마지막으로 ‘argv[]’의 시작 주소와 argc 값, return address 순서대로 스택에 저장한다.
36. 텍스트, 스크린샷이(가) 표시된 사진

    자동 생성된 설명User Memory Access
    * User Program이 접근하는 address가 유효한지 확인하는 내용을 구현하였다. 특히 system call function 호출 과정에서 User Memory access를 check하는 과정을 거친다.
    * 우선 User Memory address가 valid한지 검사하고, 만약 invalid하다면 프로그램을 종료시키는 ‘void check\_valid\_addr(void \*addr)’함수를 “userprog/syscall.c”에 구현하였다. 함수의 내부에는 Pintos 내장 함수인 ‘is\_user\_vaddr(addr)’함수를 사용하여 인자로 넘어오는 주소가 user virtual memory 영역인지 확인한다. 이를 통해 kernel virtual memory 영역에 접근하지 못하도록 한다. 또한 ‘pagedir\_get\_page()’를 사용하여 page directory에서 해당 주소가 유효하게 physical memory address와 mapped되었는지 확인할 수 있다. 또한 해당 주소가 NULL이면, 잘못된 접근이기에 이 또한 체크한다. 앞서 말한 3가지에 해당하는 경우, 프로그램을 exit(-1)을 통해 종료시킨다.
    * 앞서 선언한 ‘void check\_valid\_addr(void \*addr)’함수를 syscall\_handler()에서 사용하여 User Memory Access를 검사하게 된다. esp가 가리키는 주소가 유효한지 확인하는 역할을 하며, 각 system call에서 전달된 인자도 valid한 address에 존재하는지 확인하는 과정을 거친다. 이와 관련된 코드는 3번에서 확인할 수 있다.
37. System Calls
38. static void syscall\_handler (struct intr\_frame \*f)
39. {
40. int fd;
41. char \*buf;
42. unsigned size;
43. pid\_t pid;
44. void \*esp = f->esp;
45. check\_valid\_addr(esp);
46. int syscall\_num = \*(int \*)(esp);
47. switch(syscall\_num){
48. case SYS\_HALT:
49. halt();
50. break;
51. case SYS\_EXIT:
52. check\_valid\_addr(esp+4);
53. int status = \*(int \*)(esp + 4);
54. exit(status);
55. break;
56. case SYS\_EXEC:
57. // char \*cmd\_line;
58. check\_valid\_addr(esp+4);
59. char \*cmd\_line = (const char \*)\*(int \*)(esp+4);
60. f->eax = exec(cmd\_line);
61. break;
62. case SYS\_WAIT:
63. check\_valid\_addr(esp+4);
64. pid = (pid\_t)\*(int \*)(esp+4);
65. f->eax = wait(pid);
66. break;
67. case SYS\_READ:
68. check\_valid\_addr(esp+4);
69. check\_valid\_addr(esp+8);
70. check\_valid\_addr(esp+12);
71. fd = \*(int \*)(esp+4);
72. buf = (const char \*)\*(int \*)(esp+8);
73. size = (unsigned)\*(int \*)(esp+12);
74. f->eax = read(fd,buf,size);
75. break;
76. case SYS\_WRITE:
77. check\_valid\_addr(esp+4);
78. check\_valid\_addr(esp+8);
79. check\_valid\_addr(esp+12);
80. fd = \*(int \*)(esp+4);
81. buf = (const char \*)\*(int \*)(esp+8);
82. size = (unsigned)\*(int \*)(esp+12);
83. f->eax = write(fd,buf,size);
84. break;
85. case SYS\_MAX\_FOUR\_INT:
86. check\_valid\_addr(esp+4);
87. check\_valid\_addr(esp+8);
88. check\_valid\_addr(esp+12);
89. check\_valid\_addr(esp+16);
90. int x = \*(int \*)(esp+4),
91. y = \*(int \*)(esp+8),
92. z = \*(int \*)(esp+12),
93. w = \*(int \*)(esp+16);
94. f->eax = max\_of\_four\_int(x,y,z,w);
95. break;
96. case SYS\_FIBO:
97. check\_valid\_addr(esp+4);
98. int n = \*(int \*)(esp+4);
99. f->eax = fibonacci(n);
100. break;
101. }
102. // printf ("system call!\n");
103. // thread\_exit ();
104. }
     * 우선 void \*esp = f->esp;를 통해 system call function을 호출한 user program의 stack pointer를 참조한다. system call function이 호출되면, system call number와 argument들이 stack에 저장되어 있다. 따라서 ‘syscall\_handler()’함수는 이를 적절한 변수에 저장하여, system call funtion의 인자로 넘겨줄 수 있다.
     * Check\_valid\_addr(esp)를 통해 User program이 valid한 memory 영역을 access하는지 확인한다. 만약 유효하지 않다면, 프로그램을 종료한다.
     * “lib/syscall-nr.h”에 저장된 system call number를 토대로 switch문을 실행하여, 적절한 system call을 처리한다.
     * **텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

       자동 생성된 설명**다음은 구현된 system call funtions에 대한 설명이다.
     * ‘void halt(void)’는 Pintos 시스템을 종료하는 함수이다. Pintos 내장함수인 ‘shutdown\_power\_off()’함수를 호출하여 시스템을 종료한다.
     * **텍스트, 폰트, 스크린샷이(가) 표시된 사진

       자동 생성된 설명**
     * **텍스트, 폰트, 스크린샷이(가) 표시된 사진

       자동 생성된 설명**‘void exit(int status)’는 현재 실행 중인 process를 종료하고, 종료 상태인 status를 부모 process에 반환한다. ‘thread\_current()’함수를 통해 현재 실행중인 thread의 정보를 가져온 후, exit\_status에 종료 상태를 저장한다. 마지막으로 ‘thread\_exit()’함수를 통해 현재 thread를 종료하고, 해당 process가 사용한 자원들을 해제한다.
     * ‘pid\_t exec(const char \*cmd\_line)’는 인자로 넘어온 ‘cmd\_line’을 실행하기 위해 ‘process\_execute()’를 통해 새로운 process를 생성하고 실행한다. 추가적으로 ‘check\_valid\_addr()’를 통해 인자가 valid한 user memory 영역인지 확인한다.
     * **폰트, 텍스트, 스크린샷, 그래픽이(가) 표시된 사진

       자동 생성된 설명**
     * **텍스트, 스크린샷, 폰트이(가) 표시된 사진

       자동 생성된 설명**‘int wait(pid\_t pid)’는 자식 process가 종료될 때까지, 부모 process가 기다리도록 하는 함수이며, 자식 process가 종료되면, exit\_status를 반환한다. 내부적으로 ‘process\_wait()’함수를 호출한다.
     * ‘int read(int fd, void \*buffer, unsigned size)’는 file descriptor인 ‘fd’로부터 데이터를 읽어 ‘buffer’에 저장한다. 이번 프로젝트 경우, fd == 0인 경우(STDIN)만 처리한다. Pintos 내장함수인 ‘input\_getc()’함수를 이용하여 데이터를 한 글자씩 읽어온다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

* + **텍스트, 스크린샷, 폰트이(가) 표시된 사진

    자동 생성된 설명**‘int write(int fd, const void \*buffer, unsigned size)’는 file descriptor인 ‘fd’로 buffer의 내용을 출력한다. 이번 프로젝트 경우, fd == 1인 경우(STDOUT)만 처리한다. Pintos 내장함수인 ‘putbuf()’함수를 이용하여 buffer를 화면에 출력한다.
  + ‘exec()’함수에서 호출하는 “userprog/process.c”에 정의된 ‘tid\_t process\_execute (const char \*file\_name)’는 file\_name을 실행하기 위해 새로운 thread를 생성한다. 기존 코드에서 추가적으로 구현한 부분은, 인자로 넘어오는 ‘file\_name’의 첫 인자(실행파일명)만 tokenize하여, ‘filesys\_open(command)’를 통해 해당 파일이 존재하는지 확인 후, 파일을 open한다. 또한 ‘thread\_create(command, … )’와 같이 생성된 thread텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

    자동 생성된 설명의 이름도 실행파일명으로 저장한다.
  + 부모 process가 자식 process를 적절히 관리하기 위해 ‘struct thread’에 추가적인 변수가 필요하다. 우선 ‘int exit\_status’는 해당 process(thread)의 종료 시 반환되는 종료 상태 값을 저장한다. 이는 부모 process가 자식 process의 종료 상태를 확인할 수 있게 한다. ‘struct list child\_list’는 자식 thread를 관리하기 위한 list이다. 이 list를 순회하며 필요한 자식 thread를 찾을 수 있다. ‘struct list\_elem child\_elem’은 생성된 thread가 부모 thread의 ‘child\_list’에 추가되기 위한 list element이다. ‘struct semaphore child\_lock’은 부모 process가 자식 process가 종료될 때까지 wait하도록 synchronization을 위한 semaphore변수이다. ‘struct semaphore mem\_lock’은 자식 process가 종료된 후, 자식 process가 사용한 메모리를 해제하기 위해 사용되는 semaphore변수이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

* + 폰트, 텍스트, 스크린샷, 그래픽이(가) 표시된 사진

    자동 생성된 설명‘int process\_wait (tid\_t child\_tid)’는 ‘wait()’의 실질적인 구현이 되어있는 함수로 위와 마찬가지로 “userprog/process.c”에 정의되어 있다. 우선 부모 process의 자식 list인 ‘child\_list’에서 ‘child\_tid’에 해당하는 thread를 찾는다. 그 후, sema\_down(&child->child\_lock)을 호출하여 자식 thread가 종료될 때까지 대기한다. 자식 thread가 종료되면 ‘process\_exit()’에서 sema\_up(&t->child\_lock)되고, 부모 thread는 대기 상태에서 깨어나게 된다. 그 후, child->exit\_status를 통해 자식 process의 종료 상태를 저장한 후, 종료된 thread의 list\_elem을 부모의 child\_list에서 제거하고, 마지막으로 memory 해제를 위해 ‘sema\_up(&child->mem\_lock)’를 한다. Return 값으로 자식 process의 exit\_status를 반환한다.
  + ‘void process\_exit(void)’에서 현재 process가 종료된다면, 사용하고 있던 자원을 해제하게 된다. 이 때, 자식 process가 ‘sema\_up(&cur->child\_lock);’ 를 통해 대기하고 있던 부모 process를 깨우게 된다. 또한 ‘sema\_down(&cur->mem\_lock);‘ 을 통해 자식 process가 종료된 후, 사용하고 있던 메모리가 해제될 때 까지 자식 process가 대기하도록 한다.

1. **텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

   자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

   자동 생성된 설명**텍스트, 스크린샷, 폰트이(가) 표시된 사진

   자동 생성된 설명**텍스트, 스크린샷, 폰트이(가) 표시된 사진

   자동 생성된 설명**Additional System calls
   * 텍스트, 스크린샷, 폰트이(가) 표시된 사진

     자동 생성된 설명이번 프로젝트에서 새로 추가한 system call function으로 ‘int fibonacci(int n)’함수와 ‘int max\_of\_four\_int(int x, int y, int z, int w)’가 있다. 이를 위해 우선 “lib/syscall-nr.h”에 SYS\_FIBO와 SYS\_MAX\_FOUR\_INT를 새로운 system call number로 정의한다. 또한 “lib/user/syscall.c”에 ‘max\_of\_four\_int()’를 위한 syscall4를 새로 정의한다. 이는 인자 4개를 처리할 수 있도록 한다. 또한 앞의 두 함수를 새로 구현하여 내부적으로 적절한 system call API를 호출한다. “userprog/syscall.c”의 ‘int fibonacci(int n)’는 n번째 피보나치 수를 계산하여 반환하며, 음수를 입력하면 에러문과 함께 exit(-1)을 호출하여 프로그램을 종료한다. ‘int max\_of\_four\_int(int x, int y, int z, int w)’는 매크로 MAX(x,y) (x, y 중 더 큰 수를 반환)를 이용하여 인자로 받는 네 개의 정수 x, y, z, w 중 가장 큰 값을 반환한다.
   * 이 두 system call function을 시험하기 위해 “examples/additional.c”를 구현하고 이에 맞게 “examples/Makefile”또한 수정한다.
   1. **텍스트, 스크린샷, 폰트, 메뉴이(가) 표시된 사진

      자동 생성된 설명시험 및 평가 내용**

pintos --filesys-size=2 -p ../examples/additional -a additional -- -f -q run "additional 11 20 92 5" 을 실행했을 때의 결과

pintos --filesys-size=2 -p ../examples/additional -a additional -- -f -q run "additional 8 11 13 8" 을 실행했을 때의 결과

**텍스트, 스크린샷, 폰트, 메뉴이(가) 표시된 사진

자동 생성된 설명**