**Pintos 프로젝트 2\_2. Pintos User Program**

**(설계 프로젝트 수행 결과)**

**과목명 : [CSE4070-01] 운영체제**

**담당교수 : 서강대학교 컴퓨터공학과 박성용**

**조원 : 11조 권나영 박상욱**

**개발기간 : 2016. 11. 10. -2016. 11. 15.**

**최 종 보 고 서**

**프로젝트 제목: Pintos 프로젝트 2\_2. Pintos User Program**

**제출일: 2016. 11. 18 .**

**참여조원: 11조 권나영 박상욱**

**I. 개발 목표**

Project 2\_1에서 못다한 filesystem 관련 systemcall을 구현하는 것이 이번 프로젝트의 가장 중요한 목표이다. system call (create, remove, open, close, filesize, read, write, seek, tell)이 잘 수행될 수 있도록 만들어준다.

**II. 개발 범위 및 내용**

**가. 개발 범위**

1) project2-1에서 busywaiting방식 때문에 났던 error들을 semaphore 개념을 도입하여 wait, exec를 수정한다.

2) pintos/src/lib/user.c에 존재하는 assembly code를 수정하여 systemcall을 수행할 때 argument들이 밀려서 들어오는 것을 방지한다.

3) 이번 프로젝트에서 구현해야 할 System call(create, remove, open, close, filesize, read, write, seek, tell)을 구현하고 user program 작동시 synchronization 문제를 해결한다.

4) test case의 결과에 따라 fail이 뜨는 곳들의 error를 고친다.

**나. 개발 내용**

1) **/\*SEMAPHORE\*/**

▶ wait, exit에서 spin\_lock방식이 아닌 semaphore방식을 사용하여 동기화였다.

3) file system을 구현하기 위해 struct에 file\_descriptor와 관련된 file\_list와 관련된 부분을 추가한다

file descriptor을 관리, thread와의 connection을 쉽게 관리하기 위해 새로운 struct n\_file을 만든다.

READ/ WRITE시 Synchronization 구현

▶executable file에 관한 처리

pintos는 실행 중인 프로그램의 executable file에 다시 쓰기를 원하지 않기 때문에 file\_deny\_write함수를 적절한 상황에서 호출한다. 이는 thread가 exit할 때 (process\_Exit)에서 file\_close를 해줌으로 file\_allow\_write()함수를 호출해 다시 write가 가능해진다.

3) systemcall 구현

syscall\_ create : 파일의 이름과 초기화할 크기가 주어지면, 파일을 생성한다,

syscall\_ remove : 주어진 이름의 파일을 삭제한다

syscall\_ open : parameter로 넘어오는 명령어에 해당하는 process를 실행,

syscall\_ close : 주어진 이름의 파일을 닫아준다

syscall\_ filesize : 주어진 파일의 사이즈를 return한다

syscall\_ read : standardinput에 해당하는, 즉 fd가 0인 경우, 문자를 입력받아 버퍼에 저장, 주어진 fd의 파일, size byte만큼 buffer에 읽어 저장한다

syscall\_write : stadardoutput에 해당하는, 즉 fd가 1인 경우, 버퍼의 문자열을 출력한다. 주어진 fd의 파일에 size byt만큼 buffer에서 읽어와 그 값을 쓴다.

syscall\_seek : 현재 파일에서 읽거나 쓸 위치를 return한다

syscall\_tell : 파일의 현재 읽거나 쓰일 위치를 반환

System call 함수들의 return값은 (intr\_frame \*f)의 eax register에 저장해준다. 이 때 esp로 접근하는 argument들의 주소가 physical base를 넘지 않는지 확인해야 한다.

4) systemcall을 구현하면서 make check를 통해 fail 떴던 test들에 대해 예외처리를 한다. 총 76개 중에 create-null, read-bad-ptr, bad-read, multi-oom의 경우에 fail이 나왔다. 알맞은 예외처리를 하여 Exception을 대비한다.(방법은 아래에 서술)

**III. 추진 일정 및 개발 방법**

**가. 추진 일정**

- 11월 10일 ~ 11월 12일 : project2\_1에서 발생한 wait, exit exec 관련 오류들을 분석하고 busywaiting방식을 semaphore를 이용하는 방식으로 바꾼다. 이번 프로젝트에 사용될 structure 파악과 전반적인 흐름을 연구한다. user program이 올려질 file system을 주어진 조건에 맞게 세팅한다. Ctag를 이용하기 위해 tag를 세팅한다. 입력으로부터 들어온 명령어로부터 filename과 arguments들을 구분하여 parse하고 이를 가지고 ESP를 쌓아서 만들어준다. parse\_filename()함수, construct\_ESP()함수를 만들면서 발생한 오류를 디버깅을 통해 정정한다.

- 10월 24~25일) system call 기본적인 함수 구현, 오류정정 : hex\_dump()함수를 이용하여 filename parsing과 ESP constructing이 제대로 되었는지 확인하고 system call의 기본적인 함수들을 완성한다. Make check를 이용하여 구현했지만 실행되지 않는 문제들을 정정한다.(argument passing까지 성공) Process operation인 process\_wait함수를 만든다.

- 10월 26~28일)

**나. 개발 방법**

csprp1과 cspro2 서버에서 vi editor를 가지고 강의자료와 pintos 매뉴얼을 참고하여 개발한다.

개발 환경은 linux 컴파일 환경은 GCC이고, make file을 수정하여 project2\_1과는 다른 linker를 사용했다.

**다. 연구원 역할 분담**

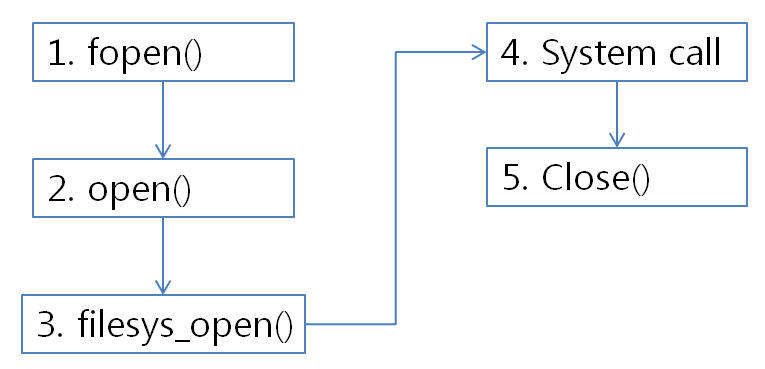
권나영 : 매뉴얼 분석, proj#2-2 ppt자료 분석, system call 구현

박상욱 : 매뉴얼 분석, proj#2-2 ppt자료 분석, busy waiting방식 semaphore로 바꿔 project2\_1의 오류 디버깅, system call 구현

**IV. 연구 결과**

**1. 합성 내용:**

**- 설계 목표에 필요한 내용을 조사 분석한 후 그들을 바탕으로 구성한 전체 프로그램 구성도**



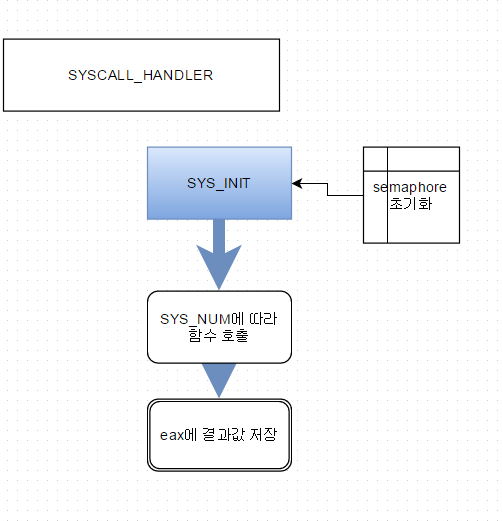
1. user program level에서 fopen()함수가 호출된다. 이번 프로젝트에서는 pintos test program에서 open()을 통해 예제 파일을 open한다.

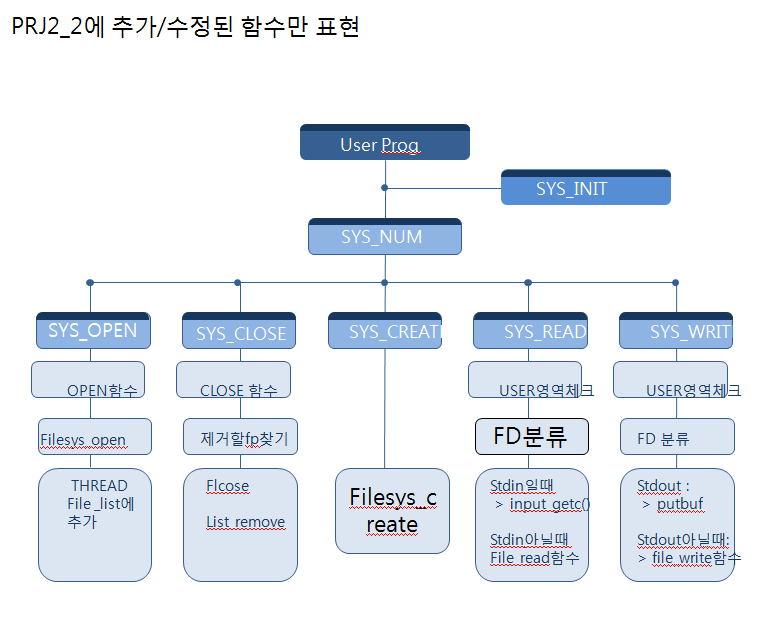
2. 호출에 의해 인터럽트가 발생하고, 이에 System call handler가 호출된다. syscall.c에서 open()함수가 호출된다.

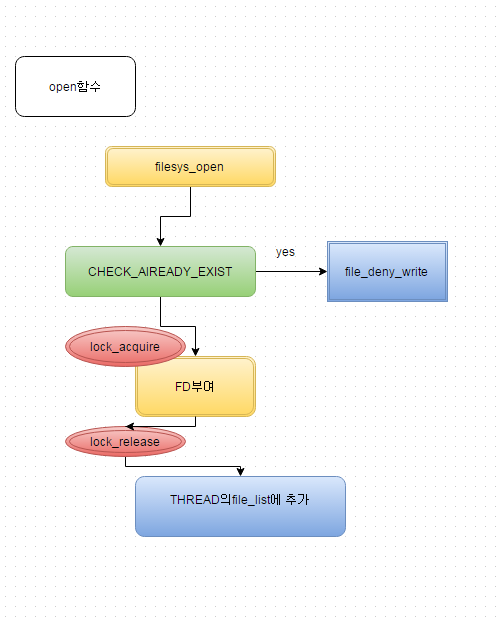
3. open함수 내부의 filesys\_open()함수가 호출된다.

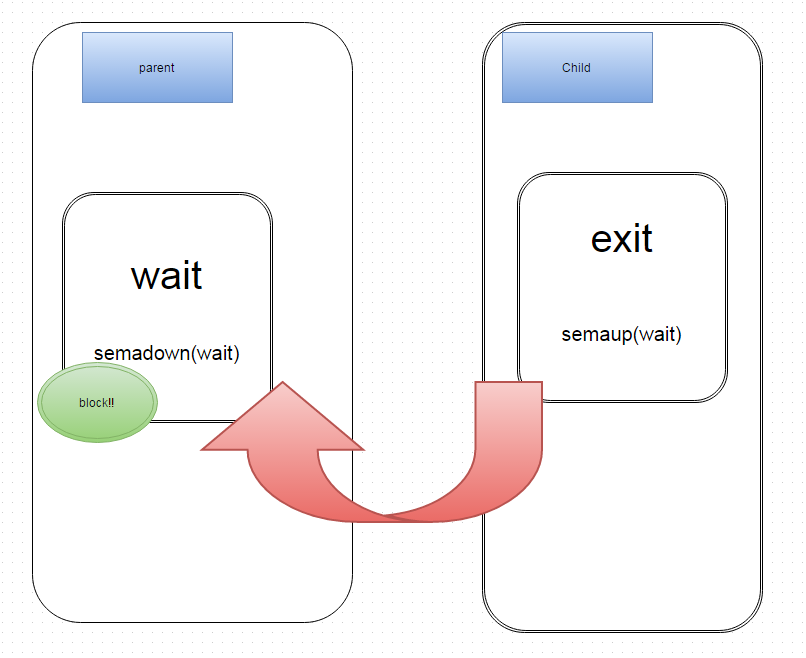
4. 이후 실행되는 file system 기반 함수(filesys\_XXXX)에 따라 open된 파일을 이용한 operation이 수행된다.

5. file을 이용한 작업이 다 끝난 후 close함수가 호출되어 file descriptor를 비롯한 변수들을 할당 해제해 준다.









**2. 제작 내용:** 개발 결과

- 구현한 컴포넌트들에 대한 역할 및 구현 방법에 관한 내용을 기술할 것. (예: 자료구조, 알고리즘).

1. Argument passing

|  |  |
| --- | --- |
| 요구사항 | 파일 관련 시스템 콜을 주어진 API를 분석하고 이를 활용하여 구현하는 것이 목표이다. |
| 조건/ 제약사항 | -내부 동기화는 없다  (즉, pintos에서 제공되는 semaphore와 같은 동기화 tool을 이용하여 동기화한다)  - file size는 creation time에 fix된다  - file data는 분할되어있지 않은 연속적인 공간에 single extend로 할당된다  - subdirectory는 없고 root directory에서만 연산한다  - file의 이름 은 제한이 있다  - kernel에서의 파일 접근은 존재하는 struct file을 통해서 가능하다  =>다음과 같은 조건 및 제약사항을 고려하여 개발한다 |
| 파일시스템  API  및 호출 구조파악 | EMB0000052c4f00  File struct 구조체  EMB0000052c4f01 |
| API  및  동기화  기술 정리 | |  |  | | --- | --- | | bool filesys\_create (const char \*name, off\_t initial\_size); | bool: 성공여부 | | 실패 경우 –이미 동일한 파일이름 있거나, allocation실패 | | 파일이름 , 초기 사이즈 | | struct file \*filesys\_open (const char \*name); | struct file\* : open한 파일  NULL :실패시 | | 파라미터로 넘겨받은 Name의 file을 찾아서 return  file이 없거나 내부 memeory 할당 실패시 NULL return | | 파일 이름 name | | bool filesys\_remove (const char \*name); | bool: 파일 삭제 성공여부 | | 파라미터로 넘겨받은 파일이름을 찾아서 삭제한다.  실패 (file이 없거나, 내부 memory allocation실패시) | |  | | void file\_close  (struct file \*); | None | | file을 닫는다 | | struct file \*file | | off\_t file\_read  (struct file \*,  void \*,  off\_t); | off\_t : 실제로 읽은 byte 수를 retun | | file 에서 size byte만큼 읽어 buffer에 저장  읽은 byte 수를 return  이는 parameter로 넘겨받은 size보다 작을수 있는데, 왜냐하면 file의 끝에 도달해서 더 이상 읽을 수없기 때문이다. file의 pos도 변한다 | | struct file \*file  void \*buffer  off\_t size | | off\_t file\_write  (struct file \*,  const void \*,  off\_t); | off\_t : write한 byte 수 | | buffer에서 size만큼 읽어서 file에 쓴다.  파일크기를 넘어서 쓸 수 없기 때문에 파라미터의 size와 return size가 다를수 있다. | | struct file \*file  void \*buffer  off\_t size | | void file\_deny\_write (struct file \*); | None | | 파일에 file이 close되거나 file\_allw\_wrtie함수가 불리어 지기 전까지 write를 금지한다. | | struct file \*file | | void file\_allow\_write (struct file \*); | None | | File의 write operation을 다시 가능하게 한다 | | struct file \*file | | void file\_seek  (struct file \*,  off\_t); | None | | file의 pos를 new\_pos로 바꿈 | | sturct file\*file, off\_t new\_pos | | off\_t file\_tell  (struct file \*); | off\_t :현재 위치 | | file->pos값을 반환 | | struct file \*file | | off\_t file\_length  (struct file \*); | off\_t : 파일의 길이 | | file의 길이를 반환한다 | | struct file \*file |   동기화함수   |  | | --- | | **void lock\_init(struct lock \*);**  =>lock를 건다  **void lock\_acquire(struct lock \*);**  =>sleep상태로 만든다 (주로 interrupt handler에서 interrupt disable과 함께 호출)  **bool lock\_try\_acquire(struct lock \*);**  => 현재 thread가 lock을 이미 가지고있지 않을 때 lock acuire를 시도해보는 것이다.  **void lock\_release(struct lock \*);**  =>현재 thread에게 소유되고있는 lock을 sema\_up을 통해 풀어준다 (INT handler에서 호출되면 안된다)  **bool lock\_held\_by\_current\_thread(const struct lock \*);**  =>lock의 소유권이 현재 thread에게 있는지를 검사하는 함수  **void sema\_init(struct semaphore \*, unsigned value);**  => semaphore의 list를 초기화하고value값으로 초기값을 지정  **void sema\_down(struct semaphore \*);**  **=>** sema값을 down 시키고 list의 waiter list에 현재 thread를 추가함  **void sema\_try\_down(struct semaphore \*);**  **=>** semaphore가 0이 아닐 때 sema를 감소시킴  **void sema\_up(struct semaphore \*);**  **=>** sema를 incre시키고, waiting list에 있는 값을 넣음 | |

file\_name이 process\_execute()함수, thread\_create()함수, load()함수를 거쳐 들어오면 이 file\_name을 filename과 argument들로 parsing하여 저장해준 다음 construct\_ESP함수로 넘겨주었다.

file\_name을 strtok\_r(tokenizer는 “ “)을 이용해 parsing했다. Strtok\_r을 사용하여 나온 첫 번째 token은 filaname(위 fime\_name과는 다름)이고 나머지 token은 argument들를 의미한다. Token의 길이를 가지고 arguments들을 동적메모리를 할당하여 argument배열을 만들어주면 parse\_filename함수는 변경된 filename과 argu\_list를 가지고 argument의 개수를 return 한다. 이때 return된 argument개수를 이용하여 construct\_ESP함수에서 esp부터 argument들을 right->left 방향으로 아래의 그림과 같은 방식으로 차례로 stack에 쌓는다.

**2\_1) 자료구조 :**

**- thread구조체**

**- file 구조체 추가**

**- 이외 동기화를 위한 semaphor**

**2\_2) 알고리즘**

**read/write함수**

Synchronization problem을 해결해야한다. Write할 때 Read하거나 read할 때 Write하는 상황, Write할 때 Write하는 상황을 방지해야한다. 이를 해결하기 위해, 강의자료에 나와있는 Readers-writers problem의 코드를 참조하여 작성하였다.

두 개의 semaphore mutex, wrt와 readcount 변수를 사용하여 Read과 Write의 동기화를 추구하였다. deny write to executable함수를 적절히 활용해야하는데 , 이미 file이 사용되고 있을 때 denying write to executable 함수를 호출할 환경을 체크하였다.

**[thread\_ wait exit함수]**

▶semaphore\_wait라는 변수를 이용해서 동기화를 구현하였다.

wait에서는

parent-child관계가 없을 때 부모가 wait하는건 아닌지 (contain이라는 지역변수로)

이미 기다리고 있는 parent-child관계는 아닌지 ( cur -> iswait라는 구조체변수로)

이미child가 죽었고 부모가 확인한건 아닌지 (cur->exit\_accepted 라는 구조체변수로)

등에 대한 예외처리로 불필요한 wait를 방지 하였다.

exit에서는 출력양식에 맞게 출력해주고 semaphore\_wait를 up해준다

▶process\_Exit (혹은 thread\_Exit) => Orphan 문제해결

비정상적인 wait와 exit으로 zombie / orphan의 문제가 발생하는 상황이 있음을 발견하였다. 이는 time\_slice가 cpu burst보다 짧아서 생길 수 있는 문제였다.

즉 , parent가 thread를 만들고 context switching이 일어나서

wait를 부르기 전에 child가 수행을 하고 이미 exit을 한 상태가 될 수 가 있다.

이를 해결하기 위해서 여러 방법을 생각해 보았다.

우선 부모 threa에서 child의 thread를 생성과 wait까지를 lock을 이용해 atomic한 process로 만들 수 있다. block시킴으로서 올바르게 정상적으로 작동이 될 것 같았다. 하지만 이는 CPU의 utilization측면에서 너무 낭비가 많을 것 같았다.

두 번째 방법으로는 dead\_list를 만드는 것이다. dead\_list라는 list 구조체를 추가하고 이를 thread에 연결하는 식으로 해결 할 수 있을 것이다. thread 구조체의 thread\_accepted라는 변수를 이용해서 spin\_lock을 걸어준다. 하지만 실제로 모든 child의 상황을 다 부모가 저장하고, 그 각각의 종료여부를 확인해야하므로 너무 불필요하게 정보를 저장해야했다.

세 번째 방법으로 semaphore통해 자식의 개수 만큼의 semaphore값을 저장하여 orphan문제를 해결할 수 있다. 이 방법을 통해 문제를 해결하였고, thread구조체에 sema\_orphan이라는 변수를 추가하였다.

정리하자면 다음 과 같다.

“sema\_orphan 이라는 변수는 내가 죽기 전에, 부모의 sema\_orphan을 down하여 아직 자식이 있으니 부모가 죽지 말고기다리라는 것을 의미한다. 나의 자식이 죽을 때 나의 sema\_orphan을 up해주어 비로소 내가 죽을 수 있을 환경을 제공해준다“

여기서 조심해야할 점은 main thread와 idle thread를 제외하여 유의미한 부모-자식간의 관계에서만 이러한 작업이 진행되어야한다. (main과 idle에서는 해당사항이 없다)

**[파일 시스템]**

**우선 syscall함수는 위에서 정리한 API를 적용해 비교적 쉽게 구현 할 수 있었다. 까다로웠던 부분이 read, write 부분과 open close 부문에서 thread와 연결해주는 부분이 비교적 어려웠다.**

**◉ syscall\_handler에서 사용할 sema\_phore 초기화**

**◉ open /close**

**-open]**

▶ file.h에 define된 FILE\_OPEN\_LMT라는 변수를 활용할 것이다

이는 #define FILE\_OPEN\_LMT 35라고 되어있고, multi-oom 에러와 관련하여 열 수 있는 파일 개수를 제한시킨다. 따라서 메모리 범람을 막는 역할을 한다.

▶동적할당(malloc)을 통해 공간을 확보하고

filesys\_open함수를 이용해서 파일을 연다.

▶여기서 file\_deny\_Write라는 함수를 사용하였다. file\_deny\_write라는 함수는 이미 FILE이 closed되었거나 이미 열려진 파일에 write를 방지하기 위해서 사용이 되었다.

▶fd를 부여하였다. sys\_fd라는 전역변수를 이용하여 작성하였는데 새로 open할 때마다 sys\_fd를 하나씩 증가 시켰고, 이 값을 만든 file 구조체의 fd에 저장하였다. 이때 lock\_aquire lock\_release를 사용하여 atomic process로 만들어서 ++가 assemble어로 변환되어 수행할 때 가져올지도 모르는 문제를 방지하였다.

▶ 이렇게 만들어진 file을 현재 thread의 file\_list에 추가하였다.

**-close]**

parameter로 넘어온 fd를 close하는 함수이다.

▶현재 thread의 file \_list에서 해당하는 file\_elem을 찾고 이를 file\_close해준다음 list\_remove이용해서 thread의 file\_list에서도 삭제해준다.

▶open시 malloc해준 영역을 free시켜준다

**◉create**

filesys\_create 함수를 이용하여 구현

**◉remove**

filesys\_remove 함수를 이용하여 구현

**◉filesize**

filesys\_filesize 함수를 이용하여 구현

**◉tell**

thread\_current()를 통해 현재 쓰레드를 찾아낸 후 그 쓰레드의 file\_list에서 fd와 같은 file\_descriptor를 가진 file을 찾아낸다. 이 file에 대하여 file\_seek함수를 호출하여 fd 파일에서 다음에 읽거나 쓰일 위치를 position의 위치로 변환하여 준다.

**◉seek**

thread\_current()를 통해 현재 쓰레드를 찾아낸 후 그 쓰레드의 file\_list에서 fd와 같은 file\_descriptor를 가진 file을 찾아낸다. 이 file에 대하여 file\_tell함수를 호출하여 fd 파일에서 다음에 읽거나 쓰일 위치를 반환하도록 한다.

**\*\*exception 해결**

bad-read문제

load에 대한 문제라는 것을 파악했고, 이는 make.config file 수정을 통해 해결

**따라서 본 프로그램은 안정성과 내구성 보건성이 좋다고 판단할 수 있다.**

2. System call

syscall\_handler : interrupt frame에 의해 얻어진 esp로 접근하여 System call number을 얻고, 해당하는 system call을 호출한다.

syscall\_halt : shutdown\_power\_off()함수를 이용하여 현재 실행중인 Pintos 실행 프로그램을 종료한다. informatio을 잃을 수 있으므로 실제로 잘 사용하지 않는다.

syscall\_exit : kernel에 status를 반환하며 현재 실행중인 thread를 종료한다. 프로세스의 부모 프로세스가 기다리고 있다면 반환될 staus는 현재의 thread의 status이다. 일반적으로 status가 0이면 성공이고 status가 0이 아니면 error를 나타낸다.

syscall\_exec : parameter로 들어온 명령어 cmd\_line을 filename으로 갖고 있는 파일을 실행한다. valid한 cmd\_line이 아닐 경우 (TID\_ERROR”–1”)을 리턴한다. process\_execute()함수로 새 thread를 생성하면서 얻은 새 process의 tid를 리턴한다.

syscall\_wait : process\_wait()함수를 이용하여 입력받은 tid를 가진 thread를 wait한다.

syscall\_read : standard read만 구현한다. fd가 0일 input\_getc()함수를 이용하여 한 바이트씩 문자를 입력받고 버퍼에 저장한다.

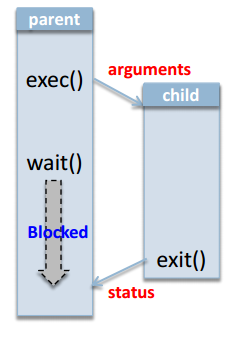
syscall\_write : standard write만 구현한다. fd가 1일 경우 경우 putbuf(buffer,size) 함수를 이용하여 버퍼의 문자열을 출력한다.

syscall\_fibonacci : 0, 1, 1, 2, 3, 5, 8, 13, 21…… 형태의 수열. 즉, 첫 번째의 값(n0)이 0이고 두 번째 값(n1)이 1일 때 이후의 항들은 이전의 두 항을 더한 값으로 만들어지는 수열을 말한다. 수열의 공식은 다음과 같다.   
fn = fn-1+ fn-2(단 , f0=0, f1=1, n=2, 3, 4, ….)

n이라는 argument를 parameter로 받았을 때, 초기값 n0=0,n1=1을 가지고 반복적으로 더해주면서 n번째 수열 값을 찾는다.

syscall\_sum\_of\_four\_integer : a,b,c,d라는 4개의 argument를 받아 이 값을 다 더한 값을 return한다.

3. Process operation

****

parent process와 child process의 전체적인 흐름은 위와 같다.

-**thread.c / thread.h**

-thread 구조체에 부모-자식에게 접근하기 위한 thread\* parent\_thread, list child\_list, child\_data \*pchild\_data를 추가하였다. 추가적으로 struct child\_data를 만들어 child와 관련된 상태를 저장한다.

**-process.c**

-process\_wait :

**-Synchronization**

-wait() 함수를 구현하는 중에 parent가 child를 기다리지 않고 종료되었을 경우 child는 orphan process가 된다. 또한 child가 wait에 status를 주지 않고 죽었을 경우 parent는 죽은 상태인지 모르므로 child process는 이름이 남아 zombie process가 된다. 이런 오류를 방지하기 위해 synchronization의 필요성이 대두된다.

**3. 시험 및 평가 내용:**

- 평가 방법에 대한 설명을 기술하라.(수행 sequence 등)

- 자신들의 결과물이 갖는 **보건 및 안정**, **생산성과 내구성**에 대하여 반드시 기술할 것.

**<MAKE CHECK 수행화면 –21개의 명령어 부분만 발췌>**

**<MAKE　GRADE 수행화면>**

**V. 기타**

- 기타 관련 내용을 기술할 것.

**1.** **연구 조원 기여도**: 기여도를 백분율로 표현할 것.

권나영 50%, 박상욱 50%

**2.** 기타 본 설계 프로젝트를 수행하면서 느낀 점을 요약하여 기술하라. 내용은 어떤 것이든 상관이 없으며, 본 프로젝트에 대한 문제점 제시 및 제안을 포함하여 자유롭게 기술할 것.

권나영 :