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

담당 교수 : 박성용

조 / 조원 : 이은지

개발 기간 : 10/05 ~ 11/2

1. **개발 목표**

* **User program을 직접 pintos 내 커널에 로드하고, 올바르게 실행할 수 있도록 알맞은 각종 System Call을 구현한다.**
* **Process와 Thread를 구분하고, Synchronization을 익혀 실행 및 대기 상태를 처리한다.**
* **만약 Kernel Memory에 접근하려는 시도를 처리하여 메모리를 안전하게 보호한다.**

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Argument Passing

명령어와 각 명령어에 알맞은 수의 인자를 Stack에 적재하여 올바르게 전달하도록 한다.

1. User Memory Access

커널 메모리를 침범하지 않도록 User Memory와 Virtual Memory를 Mapping한다.

1. System Calls

User Program을 통해 Stack에 적재된 데이터를 통하여 주로 Process 관련 System Call들을 구현한다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* **Argument Passing**
  + 커널 내 스택에 argument를 쌓는 과정 설명

프로세스를 생성 및 시작하는 process\_execute(const char\* cmd\_linde)함수가 명령어 cmd\_line을 인자로 받도록 한다. 그리고 하나의 Process는 하나의 Thread로 매핑하는 등의 One to One Mapping을 활용하였으므로, thread\_create() 내에서, 특히 이번 실습에서는 load()함수에서 command를 Parsing하여 argument를 stack에 적재한다. Parsing은 총 두 차례에 걸쳐 일어나는데

1) thread\_create()시에 첫 번째 인자로서 thread의 name이 된다. 이 경우에는 cmd\_line의 첫 단어만이 필요하므로, strtok\_r로 간단하게 공백을 구분자로 한 번만 끊어서 파싱할 수 있다.

2) load()함수 내에서 argument를 모두 추출해야하므로, 모든 인자를 띄어쓰기 기준으로 끊어 argv 버퍼에 저장하고, 추후 원활한 작업을 위해 버퍼 내 인자의 개수를 argv로 함께 전달한다. ppt에서 128바이트를 넘지 않는다고 하였으므로 128byte로 정적할당하였다.

* **User Memory Access**
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명

타 Process의 메모리를 침범하거나 Kernel Memory에 접근하려는 시도로서, 이는 자칫 프로세스가 망가지거나 Kernel에 해를 가할 수 있다.

* + Invalid memory access를 어떻게 막을 것인지 설명

이를 is\_user\_vaddr() 혹은 is\_kernel\_vaddr()와 같은 함수를 통하여 방지해야한다. 또한 메모리는 palloc\_get\_page()와 같은 함수를 통하여 안전하게 할당할 수 있다.

* **System Calls**
  + 시스템 콜의 필요성에 대한 간략한 설명

사용자가 직접 하드웨어에 접근한다면 이는 자칫 컴퓨터에 큰 손상을 끼칠 우려가 있다. 이를 방지하고자 사용자 프로그램에서 Interrupt를 보내어 커널을 거쳐 시스템의 기능을 활용해야한다.

* + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)

- **void halt(void)** : shutdown\_power\_off()를 통하여 PintOS를 자동으로 Terminate를 한다.

- **void exit(int status)** : pcb가 존재한다면, 즉 프로세스가 아직 남아있다면thread\_exit()을 호출한다.

- **void wait(pid\_t pid)** : 인자인 pid에 맞는 process\_wait()를 호출한다.

- **pid\_t sys\_exec(const char \*cmd\_line)** : process\_execute()함수를 호출하여 자식 process를 생성한다. 그리고 생성된 자식 프로세스의 프로그램이 적재될 때까지 thread\_yield()로 대기시키고, 만약 자식 프로세스가 먼저 종료되면 부모 프로세스를 비정상종료로 처리한다. 만약 자식 프로세스 적재 성공 시에는 해당 pid를 반환한다.

- **int sys\_read(int fd, void \*buffer,unsigend size)** : size에 알맞게 버퍼를 통하여 input\_getc()를 수행한다.

- **int sys\_write(int fd, const void\*buffer, unsigned size)** : size에 알맞게 버퍼를 통하여 putbuf()를 수행한다.

* + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

pid, tid 등 리턴값을 통하여 시스템 콜 정상 수행 여부 및 Process나 Child의 값을 유저에게 알려준다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**

- 10/ 5 ~ 10 / 20 : Argument Passing

- 10/ 21 ~ 10 / 24 : User Memory Access

- 10 / 25 ~ 11 / 2 : System Call Handler System. System Call Implementation

* 1. **개발 방법**
* **II. B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**
* **1. Argument Passing**

**1) thread\_create()에서의 thread name을 위한 Parsing(process.c)**

별도의 함수 필요없이 간단히 strtok\_r을 활용하여 첫 단어만 추출한다.

**2) Stack에 Argument 적재를 위한 Parsing 함수 호출**

**-> argv[] , argc 리턴(process.c)**

* **User Memory Access**

주소에 접근할 때, 그 주소가 올바른지 검사해야한다. **syscall.c**내에서 system call에서 유저 메모리에 접근할 때마다 user address가 PHYS\_BASE이하인지 체크해야한다.

모든 유저 메모리 접근은 위 read\_user, check\_user와 같은 함수를 통하기 때문에 **syscall.c**내에서 올바른 메모리 영역인지를 체크한다. 만약 할당되지 않은 page에 접근할 시 assembly line에서 page fault가 발생한다. 이를 방지하고자 잘못된 주소인 경우에는 get\_user함수로 -1로 리턴한다. 만약 커널모드에서 page fault가 발생했을 경우에는 **exception.c**의 pagefault()인터럽트 핸들러에서 eax를 -1로 설정하고 eip를 eax로 설정하여 get\_user()에서 판단가능토록 하였다.

* **System Call Handler System**

정상적인 프로그램이 수행되었다면 인터럽트가 발생하여 pintos 커널에서 이러한 인터럽트를 처리하는 핸들러로 syscall.c 내의 syscall\_handler()를 활용한다. 이는 시스템 콜의 번호에 따라 switch문을 통하여 스택 내 argument에 접근한다.

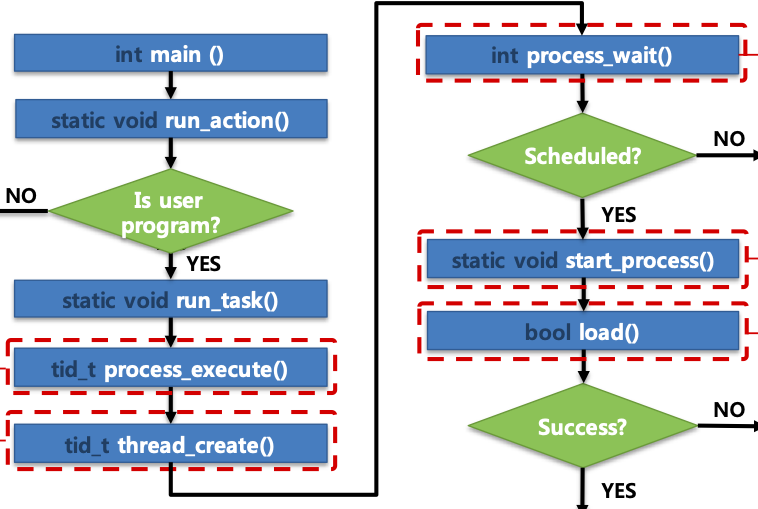
* **System Call Implementation**

각 시스템 콜 번호에 맞게 함수들이 호출되는데, 이번 프로젝트에서는 sys\_exec(), sys\_exit(), sys\_write(), sys\_read()를 우선적으로 구현하였다. 이들은 적절하게 process.c와 thread() 내의 함수를 호출하거나, semaphore 처리를 하는 핵심적인 역할을 수행하고 있다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing



load()내에서 파싱 및 적재

thread 이름 생성을 위한

Parsing

Load() 함수 내 Flow Chart

파일 open, ELF 헤더 정보를 메모리로 읽음

각 세그먼트의 가상주소 공간 위치를 읽음

User process의 페이지

테이블 생성

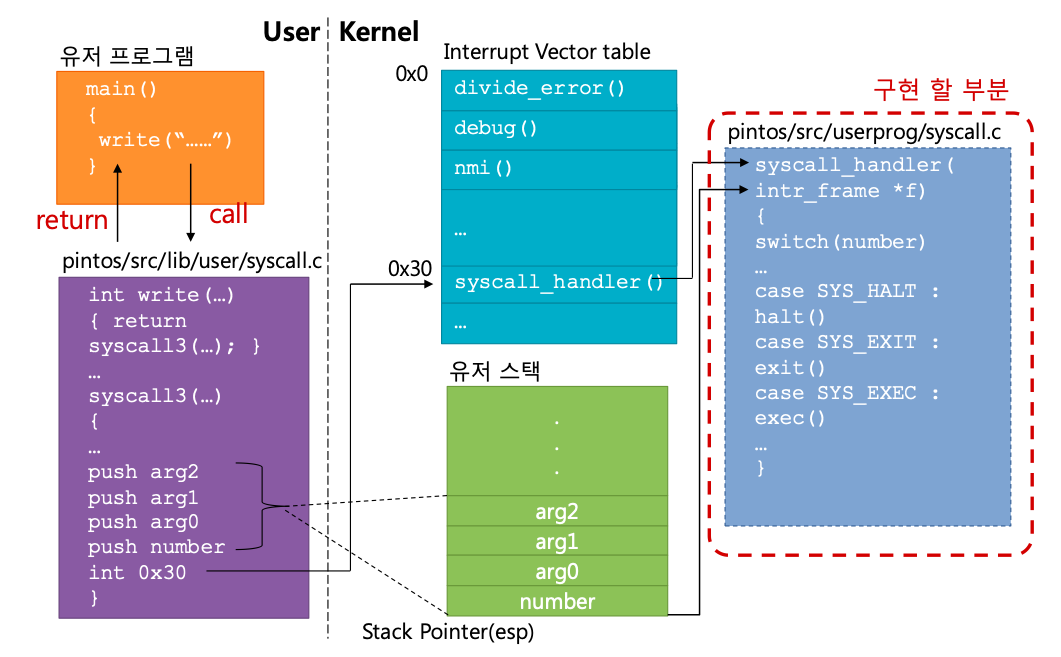
load 성공 여부 반환

스택 생성 및 초기화

파싱된 argv, argc를 스택에

적재

1. User Memory Access



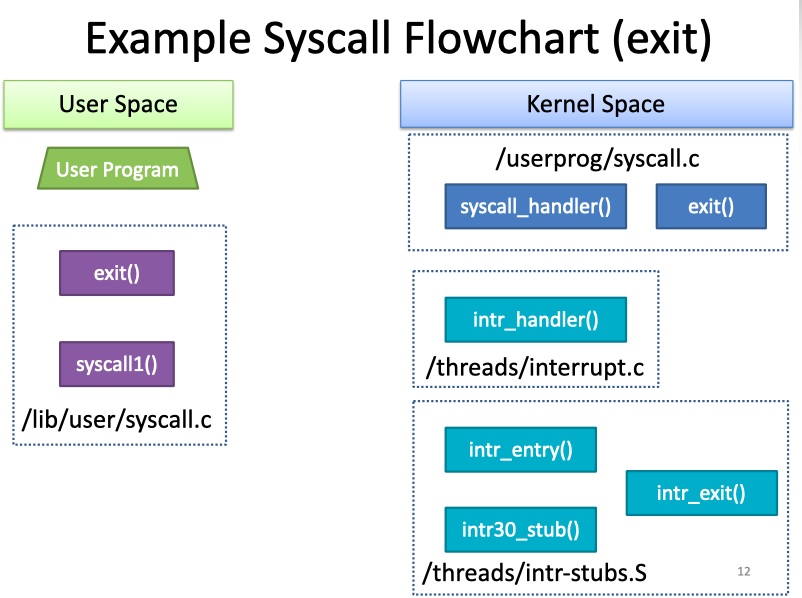
handler 내 함수에서

address가

유저영역에 있는지 확인

학인

1. System Calls



여러 시스템콜이 있지만 구체적으로 sys\_exit에 대한 플로우 차트이다.

* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Argument Passing

|  |
| --- |
| pid\_t  process\_execute (const char \*file\_name)  {  .  .  .  cmd\_copy = strtok\_r(cmd\_copy," ",&save\_ptr);  tid = thread\_create (cmd\_copy, PRI\_DEFAULT, start\_process, pcb);  .  .  .  } |

이는 위에서 언급했듯이 이름을 위한 간단한 명령어만 추출하는 과정이다.

|  |
| --- |
| bool  load (const char \*file\_name, void (\*\*eip) (void), void \*\*esp)  {  …  parse\_cmd(file\_name, argv, argc);  file = filesys\_open(argv[0]);  }  void parse\_cmd(const char \*file\_name, char \*\*argv, int \*argc){ //parse the string and save the command to the cmd  char \*fn\_copy = (char\*)palloc\_get\_page(0);//save cmd after parsing  char \*save\_ptr, \*token;  int cnt = 0;  strlcpy(fn\_copy,file\_name,PGSIZE);  for(token = strtok\_r(fn\_copy, " ",&save\_ptr);token;token = strtok\_r(NULL," ",&save\_ptr)){  argv[cnt++] = token;  }  \*argc = cnt;  }  void construct\_stack(char \*\*argv,int \*argc, void \*\* esp){  //Currently, string includes command and arguments.  //So we need to parse in this function  int i =0;  int len = 0;  void\* addrs[\*argc];  //save the address of argv  //push argv to stack  for(i = \*argc - 1;i >= 0;i--){  len = strlen(argv[i]) + 1;  \*esp -= len;  memcpy(\*esp,argv[i],len);  addrs[i] = \*esp;  }  //align word  \*esp = (void\*)((uintptr\_t)(\*esp) & 0xfffffffc);  //push NULL  \*esp -= 4;  \*(int \*)\*esp = 0;  //push argv address  for(i = \*argc-1;i>=0;i--){  \*esp -= 4;  \*(void \*\*)\*esp = addrs[i];  }  //push address of argv  \*esp -= 4;  \*((void\*\*)\*esp) = \*esp + 4;  //push argc  \*esp -= 4;  \*((int \*)\*esp) = \*argc;  //push return address  \*esp -= 4;  \*((int \*)\*esp) = 0;  } |

**stack 적재 방식**을 간단히 설명하자면 다음과 같다.

1. 전달받은 argv[argc – 1]…argv[0] 순으로 적재함과 동시에 해당 적재 주소를 별도의 배열인 addr[]에 저장한다.

2. 비트연산자를 통하여 word align을 시도한다. 핀토스의 memory unit인 page는 4kb로서 4의 배수이므로 최하위 2비트는 모두 0으로 처리하고 그보다 높은 비트는 모두 그대로 처리하여 **&연산**을 한다. (참고. 4를 이진수로 표현할 시 .)어차피 스택의 주소는 스택이 자랄수록 점차 줄어드는 추세를 보이기 때문에 이러한 방식이 적절하고 가장 간단하였다.

3. 이후 차례로 NULL, argv address, argc, 그리고 리턴 address를 삽입하고 esp를 스택에 맞게 세팅한다.

2. User Memory Access

|  |
| --- |
| ///////////////////////////////////////  /////////////Memory Access/////////////  ///////////////////////////////////////  void check\_user(const uint8\_t \*addr){  if(!is\_user\_vaddr(addr)){  sys\_exit(-1);  }  }  static int get\_user(const uint8\_t \*uaddr){  if(!((void\*)uaddr < PHYS\_BASE)){  return -1;  }  int result;  asm("movl $1f, %0; movzbl %1, %0; 1:"  : "=&a"(result): "m"(\*uaddr));  return result;  }  static int read\_user(void \*src, void \*dst, size\_t bytes){  int32\_t value;  size\_t i;  for(i=0;i < bytes;i++){  value = get\_user(src + i);  if(value == -1)//invalide memory access  sys\_exit(-1);    \*(char\*)(dst + i) = value & 0xff;  }  return (int)bytes;  } |

각 시스템콜마다 주소를 유저영역인지 검사한다. 이를 위한 검사 함수로 check\_user, get\_user, read\_user를 생성하였다. 이는 기능에 따라 약간 변화가 있다.

**check\_user :** 단순하게 해당 주소가 PHYS\_BASE 아래인지 검사하여 커널 영역을 침범하였으면 -1을 리턴하여 잘못된 접근임을 알린다.

**get\_user :** 이는 check\_user 이후 정상적인 접근일 경우에는 user memory에서 1byte를 가져온다.

**read\_user :** 정상적인 접근인 경우에는 바이트마다 get\_user를 반복하여 size만큼 바이트를 읽어 dst 버퍼에 저장하는 함수로, 안전하게 값을 가져오는 용도로 활용될 수 있다.

3. System Calls

|  |
| --- |
| syscall\_handler (struct intr\_frame \*f)  {  int syscall\_number;  read\_user(f->esp,&syscall\_number,sizeof(syscall\_number));  ASSERT(sizeof(syscall\_number == 4));    //Store the esp, which is needed in the page fault handler.  thread\_current()->current\_esp = f->esp;    switch(syscall\_number){  case SYS\_HALT: //0  {  sys\_halt();  break;  }  case SYS\_EXIT: //1  {  int exit\_code;  read\_user(f->esp + 4,&exit\_code,sizeof(exit\_code));  sys\_exit(exit\_code);  break;  }  case SYS\_EXEC: //2  {  void \*cmd\_line;  read\_user(f->esp+4,&cmd\_line,sizeof(cmd\_line));  int return\_code = sys\_exec((const char\*)cmd\_line);  f->eax = (uint32\_t) return\_code;  break;  }  case SYS\_WAIT: //3  {  pid\_t pid;  read\_user(f->esp + 4,&pid,sizeof(pid));  int ret = sys\_wait(pid);  f->eax = (uint32\_t)ret;  break;  }  case SYS\_READ: //8  {  int fd;  void \*buffer;  unsigned size;  read\_user(f->esp + 4, &fd, sizeof(fd));  read\_user(f->esp + 8, &buffer, sizeof(buffer));  read\_user(f->esp + 12, &size, sizeof(size));  int return\_code = sys\_read(fd,buffer,size);  f->eax = (uint32\_t) return\_code;  break;  }  case SYS\_WRITE: //9  {  int fd;  void \*buffer;  unsigned size;  read\_user(f->esp + 4, &fd, sizeof(fd));  read\_user(f->esp + 8, &buffer, sizeof(buffer));  read\_user(f->esp + 12, &size, sizeof(size));  int return\_code = sys\_write(fd,buffer,size);  f->eax = (uint32\_t) return\_code;  break;  }  }  } |

system call에 따라 switch문을 통하여 시스템콜에 맞게 각 함수를 호출한다. 그리고 호출하기 이전에 반드시 유저영역인지에 대한 검사가 필요하다.

|  |
| --- |
| ///////////////////////////////////////  ////////////Implementation/////////////  ///////////////////////////////////////  void sys\_halt(void){  shutdown\_power\_off();  }  void sys\_exit(int status){  printf("%s: exit(%d)\n",thread\_current()->name,status);  struct process\_control\_block \*pcb = thread\_current() -> pcb;  if(pcb != NULL)  pcb -> exitcode = status;  thread\_exit();  }  pid\_t sys\_exec(const char \*cmd\_line){  //1. Check the validity  check\_user((const uint8\_t \*)cmd\_line);  //2. Create a new process  pid\_t pid = process\_execute(cmd\_line);  if(pid == PID\_ERROR)  return pid;  //Obtain the new process  struct process\_control\_block \*child = process\_find\_child(pid);  if(child == NULL)  return PID\_ERROR;  //Wait until the new process is successfully loaded  while(child->waiting)  thread\_yield();  if(child->orphan)  return PID\_ERROR;  return pid;  }  int sys\_wait(pid\_t pid){  return process\_wait(pid);  }  int sys\_read(int fd,void \*buffer, unsigned size){  check\_user((const uint8\_t \*)buffer);  check\_user((const uint8\_t \*)buffer + size -1);  if(fd == 0){  unsigned i;  for(i = 0;i<size;i++){//STDIN  if(!input\_getc()){  sys\_exit(-1);  }  }  }  return size;  }  int sys\_write(int fd, const void \*buffer,unsigned size){  check\_user((const uint8\_t\*)buffer);  check\_user((const uint8\_t\*)buffer + size -1);  if(fd == 1){  putbuf(buffer,size);  }  return size;  } |

**1) sys\_halt()**

단순하게 shutdown\_power\_off()를 호출하면 된다.

**2) sys\_exit()**

프로세스의 pcb를 통하여 프로세스의 종료 여부를 파악하고 종료되었을 시에는 이에 따라 thread\_exit()을 호출한다.

**3) sys\_exec()**

프로세스 생성 시에 동작하는 관여하는 두 가지 주요 함수는 process\_execute()와 start\_process()이다. 따라서 syscall.c함수 외에 process.c를 추가적으로 살펴볼 필요가 있다.

|  |
| --- |
| pid\_t process\_execute (const char \*cmdline) {  /\* ... (Omitted) ... \*/  // Create a PCB, along with file\_name, and pass it into thread\_create  // so that a newly created thread can hold the PCB of process to be executed.  struct process\_control\_block \*pcb = NULL;  pcb = palloc\_get\_page(0);  // pid is not set yet. Later, in start\_process(), it will be determined.  // so we have to postpone afterward actions (such as putting 'pcb'  // alongwith (determined) 'pid' into 'child\_list'), using context switching.  pcb->pid = PID\_INITIALIZING;  pcb->cmdline = cmdline\_copy;  pcb->waiting = false;  pcb->exited = false;  pcb->orphan = false;  pcb->exitcode = -1; // undefined  // create thread, and wait until initialization inside start\_process() is complete.  tid = thread\_create (file\_name, PRI\_DEFAULT, start\_process, pcb);  if(tid == TID\_ERROR) return -1; // fail  sema\_down(&pcb->sema\_init);  // process successfully created, maintain child process list  if(pcb->pid >= 0)  list\_push\_back (&(thread\_current()->child\_list), &(pcb->elem));  return pcb->pid;  } |

위 함수 내에서 pcb 객체를 생성 및 초기화하고, thread를 생성하여 pid를 리턴해야한다. 새로운 프로세스의 로드는 process\_execute()가 실행되는 thread(부모 프로세스)가 아니라 새롭게 생성된 thread(자식 프로세스)에서 수행되는데 이것의 성공 여부에 따라 pid가 결정된다. (성공 시 tid, 실패 시 -1).따라서 부모 프로세스는 자식 쓰레드의 초기화가 완료될 때까지 대기해야한다. 이를 위해 pcb->sema\_init semaphore가 사용되는데, start\_process()가 수행되는 자식 쓰레드에서 초기화를 수행 후 실패 혹은 성공 시 release함으로써 부모프로세스가 reschedule되어 실행될 수 있도록 설계하였다.

|  |
| --- |
| void start\_process (void \*pcb\_) {  /\* ... Omitted ... \*/  struct process\_control\_block \*pcb = pcb\_;  char \*file\_name = /\* Parse and tokenize cmdline \*/;  /\* ... Omitted ... \*/  success = load (file\_name, &if\_.eip, &if\_.esp);  if (success) {  push\_arguments (cmdline\_tokens, cnt, &if\_.esp);  }  /\* Assign PCB \*/  pcb->pid = success ? (pid\_t)(t->tid) : PID\_ERROR;  t->pcb = pcb;  sema\_up(&pcb->sema\_init); // wake up sleeping in process\_execute()  if (!success) sys\_exit(-1);  /\* ... Omitted ... (jump, return from an interrupt) \*/  } |

만약 프로세스 초기화 과정이 실패한 경우(없는 파일, load 불가능, argument가 잘못되었거나, 메모리 부족, segment 형성 실패 등 다양한 요인 존재) pcb->pid = PID\_ERROR로 세팅하고 semaphore up(해제)한다. 이 경우에는 새로 생성된 thread는 sys\_exit()을 통해 자원을 모두 해제한 뒤 종료되어 부모 프로세스는 -1을 리턴한다. 만약 자식 프로세스를 성공적으로 초기화했다면, 해당 생성된 프로세스의 pcb가 부모 프로세스의 child\_list에 삽입된다. 그리고 더 정확한 구현을 위해서 추가적으로 pcb에 waiting, exited, orphan와 같은 세마포어를 추가하여 부수적으로 발생하는 exception들을 처리하였다.

**4) sys\_wait()**

wait(pid) 시스템 콜은 현재 프로세스의 자식 프로세스 중에서 pid에 해당하는 것을 찾아 종료할 때까지 대기하도록 한다. 이는 process.c의 process\_wait()함수를 활용한다.

|  |
| --- |
| int  process\_wait (tid\_t child\_tid)  {  struct process\_control\_block \*child\_pcb = process\_find\_child(child\_tid);  if(child\_pcb == NULL){//NOT found  //printf("wait() : child not found, pid = %d\n",child\_tid);  return -1;  }  if(child\_pcb->waiting){//already waiting  //printf("wait() : child found, but it is already 'waiting', pid = %d\n",child\_tid);  return -1;  }  else{  child\_pcb -> waiting = true;  }  //wait() until child terminates  //signaling this semaphore  if(!child\_pcb->exited){  sema\_down(&(child\_pcb->sema\_wait));  }  ASSERT(child\_pcb->exited == true);  //remove the child from the list  ASSERT(child\_pcb != NULL);  list\_remove(&(child\_pcb -> elem));  //return the exit code of child process  int ret = child\_pcb->exitcode;  //free pcb  palloc\_free\_page(child\_pcb);  return ret;  } |

- 현재 thread(process)의 자식 프로세스들을 child\_list라는 리스트에 저장되어 있다. 따라서 리스트를 순회하며 자식 프로세스를 찾는다. 만약 자식 리스가 없다면 -1을 리턴.

- 자식 프로세스가 이미 종료했거나, 중복 wait가 호출되는 여부는 pcb의 exited나 waiting을 보면 알 수있다. 이의 경우에도 별도로 -1 리턴 처리를 한다.

- 자식 프로세스가 아직 종료하지 않은 경우에는 현재 thread(process)는 자식 프로세스가 exit()을 호출하고 종료할 때까지 기다려야한다. 이를 알기 위해 자식 프로세스의 sema\_wait semaphore를 사용하는데, 현재 thread는 이 세마포어에서 wait하고 자식 프로세스가 종료했다는 siganal이 발생할 때까지 대기한다.

- sema\_wait에 의한 blocking에서 깨어나면, 이 상태는 자식 프로세스가 종료한 뒤(zombie process)이므로 pcb로부터 exit code를 얻을 수 있다. 이 경우에는 page를 해제한다.

#자식 프로세스가 exit()한 경우 추가 작업

- 자식 프로세스가 종료되었다는 것은 exited를 통해 할 수 있다.

- thread\_exit()을 호출하여, thread가 pintos 커널에서 종료 및 제거될 수 있도록 한다. 이과정에서 process\_exit()가 불린다.

- process\_exit()에서 먼저 점유한 메모리를 해제하고, 아직 종료되지 않은 자식 프로세스들은 orphan 처리

- sema\_wait에 시그널을 주어 자신을 기다리는 부모 프로세스가 있다면 wake up하여 부모 프로세스의 wait()가 리턴될 수 있게 한다.

- 만약 자신이 orphan process라면 , 자신의 pcb를 해제한다.

**5) sys\_read()**

- 유저 메모리의 buffer가 올바른지 양측 모두 확인한다.

- if fd == 0, stdin을 활용하여 input\_getc()함수를 통하여 size만큼 버퍼를 입력으로 채운다. if fd == 1인 경우에는 파일을 통하여 입력받는 것인데, 이는 추후 구현할 예정이다.

**6) sys\_write()**

- sys\_read()와 마찬가지로 유저 메모리의 buffer가 올바른지 양측 모두 확인한다.

- if fd == 0, stdout을 활용하여 putbuf()함수를 통하여 size만큼 버퍼를 입력으로 채운다. if fd == 1인 경우에는 파일을 통하여 출력하는 것인데, 이는 추후 구현할 예정이다.

4. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**
  1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**