+---+
| CS 330 |
| PROJECT 2: USER PROGRAMS |
| DESIGN DOCUMENT |
+---+

---- GROUP ----

>> Fill in the names and email addresses of your group members.

Geonho Koh ghkoh97@kaist.ac.kr
Hyeongseop Kim gudtjqdl789@kaist.ac.kr

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the
>> TAs, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while
>> preparing your submission, other than the Pintos documentation, course
>> text, lecture notes, and course staff.

---- REFERENCES ----

Multi-oom:https://thinkpro.tistory.com/119

ARGUMENT PASSING

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

-process.c

```
#define NUM_ARGS 128                    // 들어오는 argument의 최대 개수
#define FILE_LENTH 16                   // file_name의 최대 길이
```

---- ALGORITHMS ----

>> A2: Briefly describe how you implemented argument parsing. How do you arrange for the elements of argv[] to be in the right order? How do you avoid overflowing the stack page?

1. 먼저 file_name을 strtok_r() 함수를 이용하여 공백을 기준으로 분리한 후 포인터 배열 args를 선언하여 저장한다.

2. 스택에 argument를 쌓아주는 함수 construct_arg_stack()함수를 정의
3. construct_arg_stack 함수는 맨 뒤의 argument character부터 스택에 쌓은 후 word align한다.
4. 각 argument string이 스택에 저장된 주소를 역순으로 스택에 쌓는다. (&arg[n] -> &arg[n-1] -> …)단, &arg[0]은 NULL 값 입력.
5. 마지막으로 스택에 저장된 &arg[0] 값, argument의 개수 argc, return address(=0)를 차례로 쌓는다.

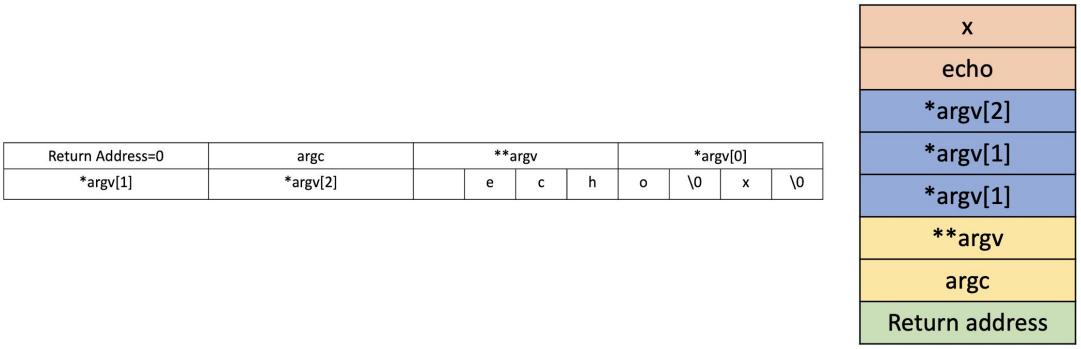

Figure A2-1 run 'echo x' 실행시 예상 스택의 구조

```
Executing 'echo x':
bffffe0  00 00 00 00 02 00 00 00-ec ff ff bf f9 ff ff bf |................|
bfffff0  fe ff ff bf 00 00 00 00-00 65 63 68 6f 00 78 00 |.........echo.x.|
```

Figure A2-2 실제 hex_dump()로 확인한 스택의 구조

*How do you arrange for the elements of argv[] to be in the right order?
strtok_r()로 자른 string을 바로 스택에 쌓는 것이 아닌, 새로운 배열을 만들어 저장 한 후에 순서에 맞게 쌓았다.

*How do you avoid overflowing the stack page?
command-line argument 가 128byte로 Limit되어 있기 때문에, 그 이상에 대한 overflow는 일어나지 않는다.

---- RATIONALE ----

>> A3: Why does Pintos implement strtok_r() but not strtok()?

strtok(char *s, const char *delimiters)
strtok_r(char *s, const char *delimiters, char **save_ptr)

strtok()은 파싱할때 정적버퍼를 사용한다.

반면, strtok_r()은 세번째 인자인 char **save_ptr을 필요로한다. 이는 정적버퍼 대신 char*로 할당된 유저에 대한 포인트를 사용하는 것이다. save_ptr은 다음처리를 위한 위치를 저장하는 Pointer이다.

strtok()과 같이 정적버퍼에 저장하면, 전역변수와 같이 data(kernel 영역)에 저장이 된다. 이렇게 kernel 영역에 저장을하면 여러 쓰레드가 동시에 돌아갈 경우 race condition의 문제가 생길 수 있다.

>> A4: In Pintos, the kernel separates commands into a executable name and arguments. In Unix-like systems, the shell does this separation. Identify at least two advantages of the Unix approach.

1 Invalid한 command를 shell에서 미리 걸러줄 수 있기 때문에 훨씬 안전하고 효율적으로 처리할 수 있다.
2 user가 자주 쓰이는 command를 저장할 수 있어 매번 직접 실행하지 않아도 된다.

SYSTEM CALLS
============

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

--thread.h

```
struct thread
{
  ...
  struct thread *parent;                  // parent thread 포인터
  struct list child_list;                 // child thread list
  struct list_elem child_elem;            // child list를 위한 list_elem

  struct semaphore exit_sema;             // syscall wait에서 child thread의
                                          // exit을 기다리기 위한 semaphore
  struct semaphore load_sema;             // syscall execute에서 child thread의
                                          // load를 기다리기 위한 semaphore

  bool load;                              // syscall execute에서  load의
                                          // 성공 여부를 확인하기 위함

  int exit_status;                        // 부모 thread가 child thread의
                                          // exit status를 확인하기 위함

  struct file *file[NUM_FILES];           // thread의 file descriptor
  int fd;                                 // file descriptors의 갯수(2로 초기화)
}

#define NUM_FILES =130                    // file의 최대 개수
```

>> B2: Describe how file descriptors are associated with open files. Are file descriptors unique within the entire OS or just within a single process?

struct thread안에 struct file *file[NUM_FILES]를 선언하여 각 single process마다 하나의 file descriptor를 가지게 했다. 이때 최대 file 의 수가 128이기 때문에 stdin, out을 고려하여 NUM_FILES=130으로 정의했다.

---- ALGORITHMS ----

>> B3: Describe your code for reading and writing user data from the kernel.

syscall에서 해당 read와 write에 대한 syscall number가 호출되면, syscall_handler에서 헤당 번호에 맞는 함수를 추가로 실행시킨다.

- sys_read
  만약 fd가 0일 때(stdin) input_getc() 함수를 활용하여 string이 끝날 때까지 한글자씩 받아 buffer에 저장한다.
  fd가 0이 아닌 경우, file_read 함수를 통해 file을 읽는다.

```
int sys_read(int fd, void *buffer, unsigned size){
  char c;
  unsigned i;
  check_valid_address(buffer, 0);
  if(fd == 0){
    for(i = 0 ; i < size ; i++){
      c = input_getc();
      if(c == '\0')
        break;
      ((char*)buffer)[i] = c;
      }
    return i-1;
   }
   struct thread *cur = thread_current();
   struct file *file = cur->file[fd];

   return file_read(file, buffer, size);
}
```

- sys_write
  만약 fd가 1일때(stdout) putbuf()를 이용해 버퍼에 입력된 내용을 저장하고, 사이즈를 리턴한다.
  fd가 0이 아닌 경우, file_write()함수를 통해 file에 쓴다. 이때 file의 deny_write을 확인하여,

file이 deny_write된 상태이면 write을 실행하지 않는다.

```
int sys_write(int fd, void *buffer, unsigned size){
 check_valid_address(buffer,0);
 if(fd == 1){
   putbuf((const char *)buffer, size);
   return size;
 }
 struct thread *cur = thread_current();
 struct file *file_d = cur->file[fd];
 if (!file_d)
    sys_exit(-1);
 if (file_d->deny_write)
    file_deny_write(cur->file[fd]);

 return file_write(file_d, buffer, size);
}
```

>> B4: Suppose a system call causes a full page (4,096 bytes) of data to be copied from user space into the kernel. What is the least and the greatest possible number of inspections of the page table (e.g. calls to pagedir_get_page()) that might result? What about for a system call that only copies 2 bytes of data? Is there room for improvement in these numbers, and how much?

byte들은 최대 2page로만 찢어질 수 있기 때문에 둘다 최소 1번, 최대 2번의 pagedir_get_page가 필요할 것이다.

>> B5: Briefly describe your implementation of the "wait" system call and how it interacts with process termination.

syscall_handler에서 wait을 실행하면, wait은 process_wait을 실행한다.

- process_wait
  주어진 tid를 가진 자식을 찾기 위해 find_child()함수를 실행한다. 이 함수는 struct thread안에 추가해준 child_list를 확인하여 해당 tid가 thread의 child_list에 있으면 자식의 struct thread 포인터를, 아니면 null을 반환한다.
  이후 sema_down()으로 child가 thread_exit()에서 sema_up()을 해줄 때까지 기다린다. child thread가 종료되면 palloc_free_page()를 해주기 전에 child_thread를 child_list에서 제거해주고, child thread의 exit_status를 저장한다. (process_exit()에 있던 palloc_free_page()를 process_wait()에서 해준다)
  마지막에 exit status를 반환해준다.

```
int process_wait (tid_t child_tid)
{
 struct thread* child = find_child(child_tid);
 int exit_status;

 if(!child){
   return -1;
 }
```

```
 sema_down(&child->exit_sema);

 list_remove(&child->child_elem);
 exit_status = child->exit_status;

 palloc_free_page(child);

 return exit_status;
}
```

>> B6: Any access to user program memory at a user-specified address can fail due to a bad pointer value. Such accesses must cause the process to be terminated. System calls are fraught with such accesses, e.g. a "write" system call requires reading the system call number from the user stack, then each of the call's three arguments, then an arbitrary amount of user memory, and any of these can fail at any point. This poses a design and error-handling problem: how do you best avoid obscuring the primary function of code in a morass of error-handling? Furthermore, when an error is detected, how do you ensure that all temporarily allocated resources (locks, buffers, etc.) are freed? In a few paragraphs, describe the strategy or strategies you adopted for managing these issues. Give an example.

위의 문제상황을 해결하기 위해, check_valid_address()함수를 만들어 주었다.

```
void check_valid_address(void *esp, int args){
 if((unsigned)esp + 4*args >= 0xc0000000 || (unsigned)esp < 0x8048000)
   sys_exit(-1);
}
```

system call에 필요한 syscall number, arguments의 정보가 모두 user stack의 범위(0x8048000 ~ 0xc0000000) 안에 존재하는지 확인한다.

*when an error is detected, how do you ensure that all temporarily allocated resources (locks, buffers, etc.) are freed?
만약 유효하지 않은 주소값이라 판단된다면, sys_exit(-1)을 통해 에러 처리를 하였다. sys_exit()을 통해 처리하면 thread_exi()t으로 가서 process_exit()에서 열려있는 파일들을 file_close()함수를 사용하여 모두 닫아 주고 pagedir_destroy()를 실행해 준다.
이와 같이 에러 발생시 thread_exit()을 통해 종료되기 때문에, allocated resource 역시 모두 제거하고 종료된다.

이 함수는 syscall_handler에서 if의 esp가 valid한지 확인해주며, 버퍼의 주소를 input으로 가지는 sys_read()와 sys_write()함수 안에서도(args = 0) 실행 해주었다.

ex) asm volatile ("movl $0xbffffffc, %%esp; movl %0, (%%esp); int $0x30"
: : "i" (SYS_EXIT));
위의 테스트 케이스에서 syscall exit은 1개의 argument를 필요로 한다. 하지만 esp + 4
의 값이 0xc0000000의 값보다 크거나 같기 때문에 syscall handler에서 exit시켜준다.

---- SYNCHRONIZATION ----

>> B7: The "exec" system call returns -1 if loading the new executable fails, so it cannot return before the new executable has completed loading. How does your code ensure this? How is the load success/failure status passed back to the thread that calls "exec"?

load가 제대로 되었는지 확인하기 위해 thread struct에 load라는 boolean 값을 추가해주었다. start_process()함수 안에서 load가 success하면 true를, fail하면 false의 값을 가지게 된다. (이는 testcase중 exec-missing(load가 fail한 case)에 대해 적용) 하지만 load가 되기전 exec이 실행되면 이를 반영할 수 없다. 따라서 load에 대해 semaphore개념을 사용해 주다.

- sys_exit()

```
tid_t
sys_exec(const char *cmd_line){
 tid_t child_pid = process_execute(cmd_line);
 struct thread *child = find_child(child_pid);
 sema_down(&child->load_sema);
 if(child->load == false)
   return -1;
 if(!child){
   return -1;
 }

 return child_pid;
}
```

- start_process()

```
start_process (void *file_name_)
{

 …

 success = load (file_name, &if_.eip, &if_.esp);


 if(success){
   construct_arg_stack(args, cnt, &if_.esp);
   thread_current()->load = true;
 }
 sema_up(&thread_current()->load_sema);

 /* If load failed, quit. */
 palloc_free_page (file_name);

 if (!success){
   list_remove (&thread_current()->child_elem);
   thread_current()->load = false;
   thread_exit();
```

```
}
```

위와 같이 child가 start_process에서 load를 다할 때까지 sema_down()을 통해 기다리고, loading 이후에는 성패 여부와 상관없이 sema_up()을 사용하여 exec을 재개하였다.

>> B8: Consider parent process P with child process C. How do you ensure proper synchronization and avoid race conditions when P calls wait(C) before C exits? After C exits? How do you ensure that all resources are freed in each case? How about when P terminates without waiting, before C exits? After C exits? Are there any special cases?

```
int
process_wait (tid_t child_tid)
{
  …
  if(!child){
    return -1;
  }

  sema_down(&child->exit_sema);

  list_remove(&child->child_elem);
  exit_status = child->exit_status;

  palloc_free_page(child);
  …
}
```

- P calls wait(C) before C exits
  sema_down()을 이용해 child thread가 process_exit에서 sema_up()을 할 때까지 parent thread 대기

- P calls wait(C) after C exits
  이미 child thread가 exit이 된 경우, parent thread의 child_list에서 제거되기 때문에 주어진 tid에 해당하는 child = NULL이 되고, wait 실패 후 -1을 반환한다.

- P terminates without waiting, before C exits
  C thread는 부모가 없는 orphan thread가 된다.

- P terminates without waiting, after C exits
  정상 작동

---- RATIONALE ----

>> B9: Why did you choose to implement access to user memory from the kernel in the way that you did?

만약 유저가 invalid한곳에 간다면, page_falut_error대신 check 함수를 통해 exit을 하게 이를통해 따라서 메모리낭비없이 문제가 생긴 스레드를 잘 정리할 수 있다.

>> B10: What advantages or disadvantages can you see to your design for file descriptors?

file descripter를 각 struct thread마다 배열로 만들어주어 thread exit시 자동으로 메모리가 free될 수 있도록 하였다.

>> B11: The default tid_t to pid_t mapping is the identity mapping. If you changed it, what advantages are there to your approach?

pintos에서는 한 process당 thread가 하나씩 있기 때문에 일대일로서 mapping 되었다.
만약에 변경을 한다면, multi-thread인 case를 위해 한개의 pid가 여러개의 tid에 연결되는 시스템을 만드는 것이 좋을 것이다.

SURVEY QUESTIONS

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want--these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

>> In your opinion, was this assignment, or any one of the three problems
>> in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave
>> you greater insight into some aspect of OS design?

>> Is there some particular fact or hint we should give students in
>> future quarters to help them solve the problems? Conversely, did you
>> find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist
>> students, either for future quarters or the remaining projects?

>> Any other comments?