# Lab2 Design Doc

## Project 2: User Programs

### 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.

My new macro definition

```
 #define ARGSNUM 128      /*the maximum number of parameters*/
#define ALIGNMENT 4      /*for word alignment*/
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x3)
/*round size to a multiple of 4 */
#define GET(ptr) (*(unsigned int *)(ptr))    /*read a word at ptr*/
#define SET(ptr,val) (*(unsigned int *)(ptr) = (unsigned int)val)    /*set a word at ptr*/
```

#### 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.How to implement argument parsing?
(1)tid_t process_execute (const char * proc_cmdline);
`process_execute` provides `proc_cmdline` ,including command and arguments string.We allocate two new pages to store `proc_cmdline` ,one to get `file_name` ,anonther as an argument passing to `start_process` .

(2)In `start_process` we allocate a new page to store `proc_cmdline` to get `file_name` .
Then we break `proc_cmdline` into words and after loading the executable,we memcpy the words to stack.In this process(as the picture showed below),we store each word's address on the stack in array `argv[]` ,and store the number of arguments in the `argc` .Also,we store the total length of the arguments into `total_len` ,which is set for alignment.

```
if_.esp = PHYS_BASE;
/*Break the command into words*/
argc = 0;
int arglen = 0;
int total_len = 0;
for (token = strtok_r (proc_cmdline, " ", &save_ptr); token != NULL;
    token = strtok_r (NULL, " ", &save_ptr))
{
  arglen = strlen(token) + 1;
  total_len += arglen;
  if_.esp -= arglen;
  memset(if_.esp,0,arglen);
  memcpy(if_.esp,token,arglen);
  argv[argc++] = if_.esp;
}
```

Then,it's time to do word alignment and push the address of each string plus a null pointer sentinel, on the stack, in right-to-left order.

```
/*Round the stack pointer down to a multiple of 4 before the first push.*/
if(total_len % 4 != 0){
  if_.esp -= (ALIGN(total_len) - total_len);
}
/*Push the address of each string plus a null pointer sentinel,on the stack,
in right-to-left order.*/
if_.esp -= sizeof(char*);
memset(if_.esp,0,sizeof(char*));
for(int num = argc-1;num >= 0;num--){
  if_.esp -= sizeof(char*);
  SET(if_.esp,argv[num]);//memcpy(if_.esp,&argv[num],sizeof(char*));
}
```

Finally,we push `argv` (the address of `argv[0]` ) and `argc` , in that order as well as a fake "return address".

```
/*push argv (the address of argv[0]) and argc, in that order.*/
void* ptr = if_.esp;
if_.esp -= sizeof(char*);
SET(if_.esp,ptr);
if_.esp -= sizeof(int);
SET(if_.esp,argc);
if_.esp -= sizeof(void*);
memset(if_.esp,0,sizeof(void*));
```

2.Way of arranging for the elements of argv[] to be in the right order.

We keep decreasing esp pointer to setup the `argv[]` elements.

```
if_.esp -= arglen;
memset(if_.esp,0,arglen);
memcpy(if_.esp,token,arglen);
argv[argc++] = if_.esp;
```

We scan through the array `argv[]` backwards(from argc-1 to 0), so that the first token we get is the last argument, the last token we get is the first argument.

```
if_.esp -= sizeof(char*);
memset(if_.esp,0,sizeof(char*));
for(int num = argc-1;num >= 0;num--){
  if_.esp -= sizeof(char*);
  SET(if_.esp,argv[num]);//memcpy(if_.esp,&argv[num],sizeof(char*));
}
```

3.How to avoid overflowing the stack page?
In thread.c,we define a function `check_magic` to detect stack overflow.

```
/*Check stack overflow*/
bool check_magic()
{
  if(thread_current()->magic != THREAD_MAGIC){
    return false;
  }
  return true;
}
```

```
/*After we set up the stack,we should check if there is stack overflow.
Use magic to check stack overflow.*/
if(!check_magic()){
  thread_exit();
}
```

My implementation didn't pre-count how much space do we need, just go through everything, make the change, like add another argv element, when necessary. After we set up the stack,we check if the magic of this thread changes to detect if there is stack overflow.If there is stack overflow,we teminate this thread.

**RATIONALE**

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

The only difference between `strtok_r` and `strtok` is that the `save_ptr` in `strtok_r` is provided by the caller. In pintos, the kernel separates commands into executable name and arguments. So we need to put the address of the arguments somewhere we can reach later.
(But my implementation doesn't take advantage of `strtok_r` ,I just allocate several new pages.This leads to a waste of space.I optimize my implementation when writing this design doc.But this design doc still corresponds to the unoptimized old version. )

> 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) It can shorten the time inside kernel.
(2) Once it can separate the commands, it can do advanced pre-processing, acting more like an interpreter not only an interface. Like passing more than 1 set of command line at a time, i.e. cd; mkdir tmp; touch test; and pipe.

## 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.

global variable:

```
 /*In filesys.h*/
struct lock file_sys_lock;
/*the lock used to protect the access to file system,this lock is initialized in function filesys_init()*/
```

new struct definition:

```
 /*In process.h*/
struct process
{
   pid_t pid;                            /*Process' identifier*/
   int exit_status;
    /*Process's exit code,which will be initialize to -1,so we don't need to change the exit
    code when error occurs.*/
   bool load_success;                    /*true for loading the executable successfully,false for
   failure*/
   bool is_alive;                        /*If this process is alive*/
   bool is_waiting;                      /*If this process is already waited by its parent.*/
   struct thread* parent;                /*Pointer which points to its parent thread*/
   struct semaphore wait_sema;   /*For implementation of system call wait*/
   struct list_elem elem;                /*make struct process linkable*/
};
/*In process.h*/
struct pfile
{
   int fd;                               /*file's indentifier*/
   struct list_elem elem;
   struct file* file;                    /*the file corresponding to this pfile*/
};
```

new member added to struct thread:

```
 /*In thread.h*/
 struct file*exec_file;                 /*the ELF of this process*/
 struct process*proc;
 /*store necessary execution information,which can be accessed by parent thread even the child process died.*/

 struct list children;                  /*list of children of this process*/
 struct semaphore exec_sema;            /*For implementation of system call exec*/
 struct list file_list;                 /*list of open files owned by this process*/
 int next_fd;                           /*the fd of the next file that will be opened by this process ,it is initialized to 2*/
```

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

In my implementation, file descriptors has a one-to-one mapping to each file
opened in a single process. The file descriptor is unique within a single process,rather than the entire OS.

## ALGORITHMS

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

Function used:
(1)check_ptr(ptr,size):Check if the address from ptr to ptr+size is valid user virtual address.If not,terminate the process.

```
static void* check_ptr(const void* ptr,size_t size)
{
  /*if vaddr is not a valid user virtual address,
  current thread exits with an error*/
  if(!is_user_vaddr(ptr) || !is_user_vaddr(ptr+size))
  {
    thread_exit();
  }
  for(size_t i = 0;i<size;i++)
  {
    if(get_user(ptr + i) == -1)
    {
      thread_exit();
    }
  }
  return (void*)ptr;
}
```

If `ptr` or `ptr+size` is not user virtual address,terminate the process.
If `get_user(ptr +i)` is equal to -1,which illustrates there is segement fault,we should terminate the process.
(2)check_write_ptr(buf,size):Check if the address from ptr to ptr+size is valid to write to.If not,terminate the process.

```c
static void *check_write_ptr(void *ptr, size_t size)
{
  if(!is_user_vaddr(ptr) || !is_user_vaddr(ptr + size)){
    thread_exit();
  }
  for(size_t i = 0; i < size; i++){
    if(!put_user(ptr + i, 0)){
      thread_exit();
    }
  }
  return ptr;
}
```

If `ptr` or `ptr+size` is not user virtual address,terminate the process.

If `put_user(ptr +i)` is flase,which illustrates there is segement fault,we should terminate the process.

(3)struct pfile* get_target_file(int fd):find the open file according to `fd` number from the process's `file_list` .If found,return the pointer pointing to the `pfile` of the target file.If not found,return NULL.

Read:

First,use `check_ptr` to check if the three arguments' address is valid.

Then,extract three parameters and use `check_write_ptr` to check if the address between `buf` and `buf + size` is valid to write to.If not,terminate the process.Also,if the `fd` of the file we are gonna to read is STDOUT_FILENO,the `fd` is not valid,we should also terminate the process.

```c
int fd = *(int*)(f->esp + sizeof(void*));
void *buf = *(void**)(f->esp + 2*sizeof(void*));
unsigned size = *(int*)(f->esp + 3*sizeof(void*));
check_write_ptr(buf,size);
if(fd == 1){
  thread_exit();
}
```

If the file'fd is STDIN_FILENO.We use `input_get` to read from the keyboard.

```c
else if(fd == 0)
{
  uint8_t*buffer = (uint8_t*)buf;
  for(size_t i = 0;i < size;i++){
    uint8_t byte = input_getc();
    put_user(buffer+i,byte);
  }
  f->eax = size;
  return;
}
```

The other circumstances,find the open file according to `fd` number from the process's `file_list` .And then use `file_read` to read the file.We need to set the return value which should be the number of the actual bytes we read from the file.If the read fails,the return value is -1.For synchronization,we should use lock to protect the critical section(file system code).

```
else
{
  struct pfile*fp = get_target_file(fd);
  if(fp == NULL){
    f->eax = -1;
    return;
  }
  lock_acquire(&file_sys_lock);
  f->eax = file_read(fp->file,buf,size);
  lock_release(&file_sys_lock);
}
```

Write:
The implementation of write is similar to read.
First,use `check_ptr` to check if the three arguments' address is valid.
Then,extract three parameters and use `check_ptr` to check if the address between `buf` and `buf + size` is valid to read from.If not,terminate the process.Also,if the `fd` of the file we are gonna to read is STDIN_FILENO,the `fd` is not valid,we should also terminate the process.

```
int fd = *(int*)(f->esp + sizeof(void*));
void *buf = *(void**)(f->esp + 2*sizeof(void*));
unsigned size = *(int*)(f->esp + 3*sizeof(void*));
check_ptr(buf,size);
if(fd == 0){
  thread_exit();
}
```

If the file'fd is STDOUT_FILENO.We use `putbuf` to write to the console.

```
else if(fd == 1)
{
  putbuf((char*)buf, size);
  f->eax = size;
}
```

The other circumstances,find the open file according to `fd` number from the process's `file_list` .And then use `file_write` to write to the file.We need to set the return value which is the number of the actual bytes we write to the file.If the write fails,the return value is -1.For synchronization,we should use lock to protect the critical section(file system code).

```
else
{
  struct pfile*fp = get_target_file(fd);
  if(fp == NULL){
    f->eax = -1;
    return;
  }
  lock_acquire(&file_sys_lock);
  f->eax = file_write(fp->file,buf,size);
  lock_release(&file_sys_lock);
}
```

> 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?

For a full page of data:
The least number is 1,if the data is contiguous and the first inspection get a page head back.We don't actually need to inspect any more, for it can contain one page of data.
The greatest number might be 4096 if it's completely not contiguous, in that case we have to check every address to ensure a valid access. When it's contiguous, the greatest number would be 2, if we get a kernel virtual address that is not a page head, we surely want to check the start pointer and the end pointer of the full page data, see if it's mapped.

For 2 bytes of data:
The least number will be 1. Like above, if we get back a kernel virtual address
that has more than 2 bytes space to the end of page.
The greatest number will also be 2. Whether it is contiguous or not,when we get back a kernel virtual address that only 1 byte far from the end of page, we have to inspect where the other byte is located.

I can't figure out how to improve these numbers.

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

If system call type is `SYS_WAIT` ,the current parent process calls `process_wait(tid)` to wait its child process to exit.And the return value will be assigned to `f—>eax` .

```
else if(args[0] == SYS_WAIT)
{
  f->eax = process_wait(args[1]);
}
```

In `process_wait` ,we first find the child process according to the number `child_tid` from current thread's `children` list.If we don't find the corresponding child process or the child process's `is_waiting` is true(parent process had called `wait` on this child process), `process_wait` immediately returns -1.In addtion to the above circumstances,if the child process is dead, `process_wait` shoud return child process's exit code.

```
int
process_wait (tid_t child_tid)
{
  struct process* pchild = find_child(child_tid);
  if(pchild == NULL || pchild->is_waiting)return -1;
  if(!pchild->is_waiting && pchild->is_alive){
    pchild->is_waiting = true;
    sema_down(&pchild->wait_sema);
    sema_init(&pchild->wait_sema,0);
    return pchild->exit_status;
  }
  else return pchild->exit_status;
}
```

If the child process is alive,we should block parent process until child process terminates.We use `semaphore` to achieve this.When we create a new process,we initialize this process's `wait_sema` to `0` . Then when we need to block the parent process,because the child process `pchild` 's `wait_sema` is `0` ,parent process will be blocked.When child process is going to die,we `sema_up` its `wait_sema` to imform its parent process of its termination.Naturally,we `sema_up` child process's `wait_sema` in `process_exit` .

```
if(cur->proc->is_waiting){
  sema_up(&cur->proc->wait_sema);
}
```

> 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.

1.how to avoid obscuring the primary function of code in a morass of error-handling:
Avoiding bad user memory access is done by checking before we really "use" the pointer and `page_fault` function.In the question `B3` ,I noted that `check_ptr(ptr,size)` is used to check if the virtual address between `ptr` and `ptr+size` is mapped,is valid to read from. `check_write_ptr(ptr,size)` is used to check if the address between `ptr` and `ptr+size` is valid user virtual address and is valid to write to. As for character string,we define a function `check_string(ptr)` to check if the character string `ptr` is valid to read.

```
static void check_string(const char*ptr)
{
  uint8_t *str = (uint8_t *)ptr;
  while(true)
  {
    if(!is_user_vaddr((void*)str))
    {
      thread_exit();
    }
    int c = get_user(str);
    if(c == -1)
    {
      thread_exit();
    }
    if(c == '\0')return;
    str++;
  }
}
```

We check each character' address to find out if it is user virtual address and if is valid to read from.If not ,we terminate the process.
So in this way,we wrap the checks for memory access into different functions,and we just need to call these functions where necessary.So we can separate the code that implements the primary function from the code that handles errors.This makes our implementation concise and clear.

Taking the bad-jump2-test( *(int*)0xC0000000 = 42; ) as an example, it's trying to write an invalid address,in `check_ptr` we find `is_user_vaddr` is false, then we will immediately terminate the process.

2.how to ensure that all temporarily allocated resources are freed when an error is detected:
Every process will call `process_exit` before it terminates.So I free the resources allocated in `process_exit`.In `process_exit`, `pagedir_destroy` will destroys page directory, freeing all the pages it references.And we will also free the file descriptors `pfile` in `file_list`.

```
/*close all the file*/
while(!list_empty(&cur->file_list))
{
  struct pfile*fp =list_entry(list_pop_front (&cur->file_list),struct pfile,elem);
  lock_acquire(&file_sys_lock);
  file_close(fp->file);
  lock_release(&file_sys_lock);
  free(fp);
}
```

As for `file_list`, we close all the open file in `file_list` and free each `pfile` pointer `fp` in `process_exit`.
Each thread' `proc` is also freed in `process_exit`.

**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"?

1.How is the load success/failure status passed back to the thread that calls "exec":
There is a member `load_success` in struct process,which represents if the load is successfully.Parent process can access child processes's member to get their `load_success` to judge if the load succeeds or not.

2.How to ensure "exec" system call won't return before the new executable has completed loading:
I use `semaphore` to achieve this.I add a new member struct semaphore `exec_sema` to struct thread.When creating a new thread,intialize `exec_sema` to `0` .

```
struct process* pchild = find_child(tid);
sema_down(&thread_current()->exec_sema);
sema_init(&thread_current()->exec_sema,0);
if(!pchild->load_success)
{
  return -1;
}
return tid;
```

In `process_execute` ,after the new child process is created,we first find the newly created process `pchild` through `tid` .Then we call `sema_down` on `exec_sema` to block the parent process.The parent process will be blocked until the new executable has completed loading.Then we need to reset `exec_sema` to `0` for the next "exec" system call.If the new executable loads successfully, `process_execute` return the child process's `pid` (equals to `tid` ).If load fails, `process_execute` returns -1.

```
if(!success){
  /*abnormally exit*/
  palloc_free_page(proc_cmdline);
  palloc_free_page(proc_copy);
  thread_current()->proc->load_success = false;
  sema_up(&thread_current()->proc->parent->exec_sema);
  thread_exit ();
}
```

In `start_process` ,if the child process' load fails,set child process's `load_success` to `false` and call `sema_up` to signal to its parent process that the load has completed.Then the child process will terminate.

```
thread_current()->proc->load_success = true;
sema_up(&thread_current()->proc->parent->exec_sema);
```

If the child process' load succeeds,set child process's `load_success` to `true` and call `sema_up` to signal to its parent process that the load has completed.

> 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?

1.How to ensure proper synchronization and avoid race conditions:
We use a `process` struct to represents each child process's status, and a
list of process( `children` ) inside parent's struct to represent all the children that
the process has. And use a semaphore `wait_sema` to ensure synchronization.C->proc's member `is_alive` represents whether C is alive or not.If C is alive,P will be blocked.Otherwise,P will immediately gets C's exit code.So there won't be race.

2.How to ensure all resources are freed in each case:

```
while(e != list_end(&cur->children)){
  struct process* pchild = list_entry(e,struct process,elem);
  pchild->parent = NULL;
  if(!pchild->is_alive){
    e = list_remove(e);
    free(pchild);
  }
  else{
    e = list_next(e);
  }
}
```

As for `children` ,we go through the list of the pointer to child processes.Because the current parent thread is going to terminate,so we change all its child processes' `parent` pointer to NULL. If the child process is dead,we can free the struct process `pchild` without any worries.If the child process is still alive,we put off the release of the resources till itself terminates.As it shows below:

```
if(cur->proc->parent == NULL){
  list_remove(&cur->proc->elem);
  free(cur->proc);
}
```

If its parent process died,there is no worry about how to pass its execution information to its parent process,so it's OK to free `proc` .
Finally all the resources we allocated will be freed.

3.Different cases:

- P calls wait(C) before C exits
  P will `sema_down` the C->proc's `wait_sema` and wait until it exits by checking the child process's existence through C->proc's member `is_alive` .(true for alive,false for dead)Then parent retrieves the child's exit status.
- P calls wait(C) after C exits
  P will find out C already exits and check it's exit status directly.
- P terminates without waiting before C exits
  C->proc' `parent` will be set to NULL,so when C terminates,its `proc` can be freed.
- P terminates after C exits
  When P terminates,all its dead child thread's `proc` will be freed,so C's `proc` will be freed.

**RATIONALE**

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

I choose the second method: check only that a user pointer points below `PHYS_BASE` , then dereference it.
Reasons:
(1)This technique is normally faster because it takes advantage of the processor's MMU.
(2)Use `get_user` and `put_user` ,we can seperate the check for reading from user memery from the check for writing to user memory and define different wrapped function.For me,I think it's more clear logically.

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

Advantages:
(1) The definition is simple and clear.Creating a new file descriptor doesn't cost much space.

```
struct pfile
{
    int fd;
    struct list_elem elem;
    struct file* file;
};
```

Disadvantages:
(1)Thread-struct's can consume a lot of space,for we add a list of open files to thread structure.
(2) The inherits of open files opened by a parent require extra effort to be
implement.

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

I didn't change it. I think the identity mapping works well.