CS342: Operating Systems Lab
Department of Computer Science and Engineering,
Indian Institute of Technology, Guwahati
North Guwahati, Assam 781 039 Exercise UP-04

OS Lessons: Exec(), Wait() and Denying Writes to Executables
Rating: Hard
Last update: 12 Sept 2017

Please do not start on this exercise before you have successfully completed Exercise UP-02. Attempting to progress without a good understanding of the solutions needed in Exercise UP-02 is likely to be expensive on your time and efforts. It is also a good idea to complete Exercise UP-03 first.

Tasks for the Exercise:

In this exercise, we work on system calls `exec()` and `wait()`. System call `exec()` specifications may be modelled on function `process_execute`() that was settled in an earlier exercise. However, you must also make sure that your implementation does not report a successful completion of the system call until after the child process/thread is at a stage where it will run as a process.

The latter requirement is not trivial. A new process can fail to reach a successful birth for many reasons. Thus, a successful birth should be report at the completion of all stages; and, not at the start when memory for `struct thread` is allocated.

PintDoc suggests that `wait()` implementation is a difficult task. The remark about the difficulty is not without its merit.

Described below is a possible design suggestion that you may use for these two system calls. It is not a praise-worthy plan but it worked for us. We used a set of 4 arrays indexed by thread identifier. For each thread, the array-set records data that is not conveniently recorded in `struct thread`. Example of information that is inconvenient to record in `struct thread` is information that is needed even if the thread is not created successfully or data that is needed after the thread has exited.

Some details of our implementation of these 4 arrays is given below but we do expect that the keen students will improve on these and construct better data-structures to record data about each thread that PintOS kernel needs outside the life-span of a thread. `Struct thread` is a convenient data-structure to hold data needed during the active life-span of a thread.

1. One of the arrays we used provides mapping from thread-identifier (`tid`) to the matching thread's `struct thread`.
2. The second array was used to record the successful birth of thread `tid`. A successful birth is achieved only when the process's program code has been loaded. A `tid` is allocated as soon as space is allocated for data-structure `struct thread`. PintDoc requires that `exec()` does not return thread's `tid` before the thread is properly established.
3. The third array is used to park `exit-status` of a thread for system call `wait()` from the thread's parent.

4. Our final array helps in managing accesses to these data-structures during and after the thread's THREAD_DYING state.

Swiss Army Knife:

`Struct thread` supports the creation of a new thread by providing a fake *past* to the new thread. This cleverly created past of the thread helps the thread to join `ready_list` and be able to run when scheduled in the same way as the other threads.

In implementing `exec()`, the students need to understand the structure in some details. Indeed, this complexity is the reason, we delayed the implementation of `exec()` and `wait()` as the last exercise of the project. To help you understand the details, we list below function `thread_create()` from file `threads/thread.c`:

```
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  struct thread *t;
  struct kernel_thread_frame *kf;
  struct switch_entry_frame *ef;
  struct switch_threads_frame *sf;
  tid_t tid;
  enum intr_level old_level;

  ASSERT (function != NULL);

  /* Allocate thread. */
  t = palloc_get_page (PAL_ZERO);
  if (t == NULL)
    return TID_ERROR;

  /* Initialize thread. */
  init_thread (t, name, priority);
  tid = t->tid = allocate_tid ();

  /* Prepare thread for first run by initializing its stack.
     Do this atomically so intermediate values for the 'stack'
     member cannot be observed. */
  old_level = intr_disable ();

  /* Stack frame for kernel_thread(). */
  kf = alloc_frame (t, sizeof *kf);
  kf->eip = NULL;
  kf->function = function;
  kf->aux = aux;

  /* Stack frame for switch_entry(). */
  ef = alloc_frame (t, sizeof *ef);
  ef->eip = (void (*) (void)) kernel_thread;

  /* Stack frame for switch_threads(). */
  sf = alloc_frame (t, sizeof *sf);
```

```
  sf->eip = switch_entry;
  sf->ebp = 0;

  intr_set_level (old_level);

  /* Add to run queue. */
  thread_unblock (t);

  return tid;
}
```

A newly created thread, begins its life by executing function `kernel_thread()`. The function is copied below to aid our explanation. Note in the code above, how a call to function `kernel_thread()` is included in the code. This is not the way your first programming course taught you! The function run will be initiated by the PintOS scheduler.

```
/* Function used as the basis for a kernel thread. */
static void
kernel_thread (thread_func *function, void *aux)
{
  ASSERT (function != NULL);

  intr_enable ();  /* The scheduler runs with interrupts off. */
  function (aux);  /* Execute the thread function. */
  thread_exit ();  /*If function() returns, kill the thread. */
}
```

You need to recognize that the function finds its arguments on a stack created previously during `thread_create()` execution. The arguments were inserted by the parent thread! Argument `aux` is really the command line specifying the user program with arguments to be run in the process being assembled. And, argument `function`, is function `start_process()` – different from the user program you might have expected!

Function `start_process()` is responsible to load the user program and setup user-program's initial stack. All the needed information is in argument `aux`. We discussed this issue in exercise UP-01.

Function `kernel_thread()` is run as a new entity (child thread) separate from the thread that called function `thread_create()` . The separation occurred near the end of the function `thread_create()` as should be noted through the code:

```
  /* Add to run queue. */
  thread_unblock (t);
```

For the sake of completeness we say, all context switch from a running thread to a new running thread occur through function `switch_threads()`. Thus, the thread exiting status THREAD_RUNNING and the thread entering status THREAD_RUNNING seamlessly execute the same code in function `switch_threads()`.

The first switch for a thread is special, as it has no past to resume from. This void was filled by a fake past as we discussed earlier. Function `thread_create()` provides frame `switch_entry()` and it is no surprise that it causes the thread to "return-from-call" to the start of function `kernel_thread()`. This is the function every thread runs at their birth.

It may be interesting and useful to read the appendix before reading this section again. But this is a magic trick every good computer student must learn and master. Appendix *A.2.3 Thread* Switching is the authentic source of information for PintOS lovers.

Now Enjoy Coding the Final Part of Project:

The specifications for system calls `exec()` and `wait()` are near replica of functions `process_execute()` and `process_wait()` in file `userprog/process.c`. But, the differences are important too.

We expect (and recommend) that you work on this implementation in three phases. In Phase 1, focus on developing code to correctly implement system call `exec()`.

In the Phase2, include system call `wait()` into your goals.

In the final phase, you work on the specifications set in Section *3.3.5 Denying Writes to Executables* of PintDoc.

The success of each phase, as determined by the success status of command `make check`, is given below to help you monitor your progress.

Status of our "*make* check":

After full implementation of *exec():*

```
[vmm@progsrv build]$ make check
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
```

```
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
FAIL tests/userprog/exec-once
FAIL tests/userprog/exec-arg
FAIL tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
FAIL tests/userprog/wait-simple
FAIL tests/userprog/wait-twice
FAIL tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
FAIL tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
FAIL tests/userprog/rox-simple
FAIL tests/userprog/rox-child
FAIL tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
FAIL tests/userprog/no-vm/multi-oom
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
FAIL tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
FAIL tests/filesys/base/syn-write
13 of 76 tests failed.
make: *** [check] Error 1
```

On completion of `exec()` and `wait()` system calls:

Our success status improved to just 4 FAILs:

```
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
FAIL tests/userprog/rox-simple
FAIL tests/userprog/rox-child
FAIL tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/userprog/no-vm/multi-oom
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
FAIL tests/filesys/base/syn-write
4 of 76 tests failed.
make: *** [check] Error 1
```

On completion of all three phases:

And, finally after successful implementation of *Denying Writes to Executables* we could pass all tests. This part does require a lot of thought and several score lines of kernel code. We provide necessary support below.

The output below is also a confirmation that if a student meticulously follows the instructions, all tests in *User Program* project can be successfully completed.

In brief, the implementation requires that for each executable file that has been loaded in one or more processes running on the system, we maintain a count of the processes that have loaded the executable file. The denial of write obligation on the executable file stays till the last process loaded with the file has terminated.

A smart student would have already sensed that it requires some smart programming as we cannot know which process loaded with this executable file will be the last to terminate. It also stands to reason that the process that loaded the executable file first (and hence initiates the constraint “deny write on the file”), is likely to be among the early processes to finish ahead of those who loaded the same file at later time. In a careless implementation, the constraint "*deny write on file*" may be lifted inadvertently as soon as this (first) process terminates even though there may be other processes running file code are still active in the system.

The problem described in the last paragraph is not the only tricky issue that you need to handle. You also need to understand that not all threads are subject to a `wait()` call from their parent thread. That is, `wait()` is not a reliable indicator of the termination of a process. However, all resources given to a thread must be returned on its completion. That is, to avoid resource leakage every page carrying a `struct thread` data-structure needs to be freed by calling `palloc_free_page()`. Likewise, every open files must be closed. If we fail to deallocate all resources, the kernel shall degrades over time. Our resource deallocation plan cannot rely on function `wait()` to free resources.

The test `tests/userprog/no-vm/multi-oom` only succeeded when the implementation supported 2040 threads each with ability to host 128 open files simultaneously. These numbers were used to set the sizes for the arrays in our implementation of the project.

```
[vmm@progsrv ~]$ cd pintos/src/userprog/build/
[vmm@progsrv build]$ make check
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
```

```
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/userprog/no-vm/multi-oom
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 76 tests passed.
```


PS: A comment

The extension described above was completed separate from the extensions to project `Threads`. (Actually, project `UserProg` was done first). In August/Sept 2017, attempts were made to combine the code from these two projects. The combined code caused problems with test `multi-oom` as `struct thread` does not have enough space for stack to accommodate all data elements and the kernel stack.

We could accommodate only 99 files in a process. This causes some other tests to fail.

It was decided to separate allocation for pointers `strcut file *` into a separate `calloc()` area.

After these changes, time (6 minutes) for test `multi-oom` is tight when using simulator `bochs`. You may need to adjust for this time limitation by using a longer time duration for this test.

To our utter surprise, setting the limit for (open) files at 63 helps pass all tests.

Appendix: A Visit to PintOS Corporation

PintOS Corporation (PintCorp) has many employees and there is a significant turn-over of these employees. New employees are contracted and old leave PintCorp regularly.

The company has a famous coffee house, where employees come often during their work to rest, relax and of course to drink. Talking business here is an absolute no-no. The coffee house has three doors. One of the door at the front is the entrance for the new employees. Each new employee gets to enjoy a coffee break before work begins. There is a separate door to let the employees who were away come back to work. They too get to enjoy their coffee before resuming duties. These two doors are for entry only – no one is allowed out of these doors. Employees at work enter and exit coffee house through the third door.

Employees spend as much time as they like in the coffee house. They can come in any time and leave any time. An employee, who is not new to the company, goes to his work-desk and resumes work diligently. The employees work till the work is complete or they want to have a coffee or they need to go out of the Corporate area.

The arrangements for the new employee are practical. A new employee does not have a desk to work from. They are given directions to the desk store. That is where they report to start their work at PintCorp. The store gives the new employee a desk to work from. The new employee sets their desk up and begin working.

PintCorp has no manager! Every employee follows the rules and work with the corporation till the work is finished. The existing employees are able to hire new employees into PintCorp. A final fact about PintCorp is that no more than one employee ever works at a time. Others are either out of office or having a coffee break.

The employee who hires a new employee may wait for the hired employee to finish work before returning to work. But, some employees continue to work without waiting for the hired employees to finish their assigned works.

If you have read the story carefully, you know that PintCorp calls its employees threads. Each has a number `tid`. The coffee shop is called `THREAD_READY`. The working employee is termed `THREAD_RUNNING`. And, those away are known as `THREAD_BLOCKED`.

Code Written

Original PintOS versus Code implementing only project User Program (but not project Threads)

File	Original	After User Program
`threads/thread.c (thread.h)`	587 (141)	709 (166)
`userprog/exception.c`	161	163
`userprog/process.c (process.h)`	465 (11)	657 (14)
`userprog/syscall.c (syscall.h)`	20 (6)	332 (7)

Original PintOS versus combined projects Threads and User Program

Files	Original	Combined
`devices/timer.c (timer.h)`	255 (29)	328 (34)
`threads/synch.c (synch.h)`	338 (51)	571 (56)
`threads/thread.c (thread.h)`	587 (141)	991 (193)
`userprog/process.c (process.h)`	465 (11)	667 (14)
`userprog/syscall.c (syscall.h)`	20 (6)	335 (7)
`userprog/exception.c`	161	163

Contributing Authors:

Vishv Malhotra, Gautam Barua, Rashmi Dutta Baruah