

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: 20 February 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 = pallocc_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);
```

```

90     sf->eip = switch_entry;
91     sf->ebp = 0;
92
93     intr_set_level (old_level);
94
95     /* Add to run queue. */
96     thread_unblock (t);
97
98     return tid;
99 }

```

100

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

```

105 /* Function used as the basis for a kernel thread. */
106 static void
107 kernel_thread (thread_func *function, void *aux)
108 {
109     ASSERT (function != NULL);
110
111     intr_enable (); /* The scheduler runs with interrupts off. */
112     function (aux); /* Execute the thread function. */
113     thread_exit (); /*If function() returns, kill the thread. */
114 }

```

115

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

121 Function `start_process()` is responsible to load the user program and setup user-program's  
 122 initial stack. All the needed information is in argument `aux`. We discussed this issue in exercise UP-  
 123 01.

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

```

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

```

129

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

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

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

## 141 Now Enjoy Coding the Final Part of Project:

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

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

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

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

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

152

## 153 Status of our “*make check*”:

154 After full implementation of `exec()` :

```
155 [vmm@progsrv build]$ make check
156 pass tests/userprog/args-none
157 pass tests/userprog/args-single
158 pass tests/userprog/args-multiple
159 pass tests/userprog/args-many
160 pass tests/userprog/args-dbl-space
161 pass tests/userprog/sc-bad-sp
162 pass tests/userprog/sc-bad-arg
163 pass tests/userprog/sc-boundary
164 pass tests/userprog/sc-boundary-2
165 pass tests/userprog/halt
166 pass tests/userprog/exit
167 pass tests/userprog/create-normal
168 pass tests/userprog/create-empty
169 pass tests/userprog/create-null
170 pass tests/userprog/create-bad-ptr
171 pass tests/userprog/create-long
172 pass tests/userprog/create-exists
173 pass tests/userprog/create-bound
174 pass tests/userprog/open-normal
175 pass tests/userprog/open-missing
176 pass tests/userprog/open-boundary
177 pass tests/userprog/open-empty
```

178 pass tests/userprog/open-null  
179 pass tests/userprog/open-bad-ptr  
180 pass tests/userprog/open-twice  
181 pass tests/userprog/close-normal  
182 pass tests/userprog/close-twice  
183 pass tests/userprog/close-stdin  
184 pass tests/userprog/close-stdout  
185 pass tests/userprog/close-bad-fd  
186 pass tests/userprog/read-normal  
187 pass tests/userprog/read-bad-ptr  
188 pass tests/userprog/read-boundary  
189 pass tests/userprog/read-zero  
190 pass tests/userprog/read-stdout  
191 pass tests/userprog/read-bad-fd  
192 pass tests/userprog/write-normal  
193 pass tests/userprog/write-bad-ptr  
194 pass tests/userprog/write-boundary  
195 pass tests/userprog/write-zero  
196 pass tests/userprog/write-stdin  
197 pass tests/userprog/write-bad-fd  
198 FAIL tests/userprog/exec-once  
199 FAIL tests/userprog/exec-arg  
200 FAIL tests/userprog/exec-multiple  
201 pass tests/userprog/exec-missing  
202 pass tests/userprog/exec-bad-ptr  
203 FAIL tests/userprog/wait-simple  
204 FAIL tests/userprog/wait-twice  
205 FAIL tests/userprog/wait-killed  
206 pass tests/userprog/wait-bad-pid  
207 FAIL tests/userprog/multi-recurse  
208 pass tests/userprog/multi-child-fd  
209 FAIL tests/userprog/rox-simple  
210 FAIL tests/userprog/rox-child  
211 FAIL tests/userprog/rox-multichild  
212 pass tests/userprog/bad-read  
213 pass tests/userprog/bad-write  
214 pass tests/userprog/bad-read2  
215 pass tests/userprog/bad-write2  
216 pass tests/userprog/bad-jump  
217 pass tests/userprog/bad-jump2  
218 FAIL tests/userprog/no-vm/multi-oom  
219 pass tests/filesys/base/lg-create  
220 pass tests/filesys/base/lg-full  
221 pass tests/filesys/base/lg-random  
222 pass tests/filesys/base/lg-seq-block  
223 pass tests/filesys/base/lg-seq-random  
224 pass tests/filesys/base/sm-create  
225 pass tests/filesys/base/sm-full  
226 pass tests/filesys/base/sm-random  
227 pass tests/filesys/base/sm-seq-block  
228 pass tests/filesys/base/sm-seq-random  
229 FAIL tests/filesys/base/syn-read  
230 pass tests/filesys/base/syn-remove  
231 FAIL tests/filesys/base/syn-write  
232 13 of 76 tests failed.  
233 make: \*\*\* [check] Error 1

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

235 Our success status improved to just 4 FAILs:

```
236 pass tests/userprog/wait-twice
237 pass tests/userprog/wait-killed
238 pass tests/userprog/wait-bad-pid
239 pass tests/userprog/multi-recurse
240 pass tests/userprog/multi-child-fd
241 FAIL tests/userprog/rox-simple
242 FAIL tests/userprog/rox-child
243 FAIL tests/userprog/rox-multichild
244 pass tests/userprog/bad-read
245 pass tests/userprog/bad-write
246 pass tests/userprog/bad-read2
247 pass tests/userprog/bad-write2
248 pass tests/userprog/bad-jump
249 pass tests/userprog/bad-jump2
250 pass tests/userprog/no-vm/multi-oom
251 pass tests/filesys/base/lg-create
252 pass tests/filesys/base/lg-full
253 pass tests/filesys/base/lg-random
254 pass tests/filesys/base/lg-seq-block
255 pass tests/filesys/base/lg-seq-random
256 pass tests/filesys/base/sm-create
257 pass tests/filesys/base/sm-full
258 pass tests/filesys/base/sm-random
259 pass tests/filesys/base/sm-seq-block
260 pass tests/filesys/base/sm-seq-random
261 pass tests/filesys/base/syn-read
262 pass tests/filesys/base/syn-remove
263 FAIL tests/filesys/base/syn-write
264 4 of 76 tests failed.
265 make: *** [check] Error 1
```

266

267 On completion of all three phases:

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

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

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

277 A smart student would have already sensed that it requires some smart programming as we cannot  
278 know which process loaded with this executable file will be the last to terminate. It also stands to  
279 reason that the process that loaded the executable file first (and hence initiates the constraint “deny  
280 write on the file”), is likely to be among the early processes to finish ahead of those who loaded the

281 same file at later time. In a careless implementation, the constraint “*deny write on file*” may be lifted  
282 inadvertently as soon as this (first) process terminates even though there may be other processes  
283 running file code are still active in the system.

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

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

```
294 [vmm@progsrv ~]$ cd pintos/src/userprog/build/  
295 [vmm@progsrv build]$ make check  
296 pass tests/userprog/args-none  
297 pass tests/userprog/args-single  
298 pass tests/userprog/args-multiple  
299 pass tests/userprog/args-many  
300 pass tests/userprog/args-dbl-space  
301 pass tests/userprog/sc-bad-sp  
302 pass tests/userprog/sc-bad-arg  
303 pass tests/userprog/sc-boundary  
304 pass tests/userprog/sc-boundary-2  
305 pass tests/userprog/halt  
306 pass tests/userprog/exit  
307 pass tests/userprog/create-normal  
308 pass tests/userprog/create-empty  
309 pass tests/userprog/create-null  
310 pass tests/userprog/create-bad-ptr  
311 pass tests/userprog/create-long  
312 pass tests/userprog/create-exists  
313 pass tests/userprog/create-bound  
314 pass tests/userprog/open-normal  
315 pass tests/userprog/open-missing  
316 pass tests/userprog/open-boundary  
317 pass tests/userprog/open-empty  
318 pass tests/userprog/open-null  
319 pass tests/userprog/open-bad-ptr  
320 pass tests/userprog/open-twice  
321 pass tests/userprog/close-normal  
322 pass tests/userprog/close-twice  
323 pass tests/userprog/close-stdin  
324 pass tests/userprog/close-stdout  
325 pass tests/userprog/close-bad-fd  
326 pass tests/userprog/read-normal  
327 pass tests/userprog/read-bad-ptr  
328 pass tests/userprog/read-boundary  
329 pass tests/userprog/read-zero  
330 pass tests/userprog/read-stdout  
331 pass tests/userprog/read-bad-fd
```

332 pass tests/userprog/write-normal  
333 pass tests/userprog/write-bad-ptr  
334 pass tests/userprog/write-boundary  
335 pass tests/userprog/write-zero  
336 pass tests/userprog/write-stdin  
337 pass tests/userprog/write-bad-fd  
338 pass tests/userprog/exec-once  
339 pass tests/userprog/exec-arg  
340 pass tests/userprog/exec-multiple  
341 pass tests/userprog/exec-missing  
342 pass tests/userprog/exec-bad-ptr  
343 pass tests/userprog/wait-simple  
344 pass tests/userprog/wait-twice  
345 pass tests/userprog/wait-killed  
346 pass tests/userprog/wait-bad-pid  
347 pass tests/userprog/multi-recurse  
348 pass tests/userprog/multi-child-fd  
349 pass tests/userprog/rox-simple  
350 pass tests/userprog/rox-child  
351 pass tests/userprog/rox-multichild  
352 pass tests/userprog/bad-read  
353 pass tests/userprog/bad-write  
354 pass tests/userprog/bad-read2  
355 pass tests/userprog/bad-write2  
356 pass tests/userprog/bad-jump  
357 pass tests/userprog/bad-jump2  
358 pass tests/userprog/no-vm/multi-oom  
359 pass tests/filesys/base/lg-create  
360 pass tests/filesys/base/lg-full  
361 pass tests/filesys/base/lg-random  
362 pass tests/filesys/base/lg-seq-block  
363 pass tests/filesys/base/lg-seq-random  
364 pass tests/filesys/base/sm-create  
365 pass tests/filesys/base/sm-full  
366 pass tests/filesys/base/sm-random  
367 pass tests/filesys/base/sm-seq-block  
368 pass tests/filesys/base/sm-seq-random  
369 pass tests/filesys/base/syn-read  
370 pass tests/filesys/base/syn-remove  
371 pass tests/filesys/base/syn-write  
372 All 76 tests passed.  
373  
374



## 375 Appendix: A Visit to PintOS Corporation

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

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

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

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

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

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

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

403

404 **Contributing Authors:**

405 Vishv Malhotra, Gautam Barua, Rashmi Dutta Baruah