Implementation requirements

TABLE OF CONTENTS

New syscalls

For this project, you will need to implement the following new system calls:

pthread syscalls

`tid_t sys_pthread_create(stub_fun sfun, pthread_fun tfun, const void* arg)`

Creates a new user thread running stub function `sfun`, with arguments `tfun` and `arg`. Returns TID of created thread, or `TID_ERROR` if allocation failed.

`void sys_pthread_exit(void) NO RETURN`

Terminates the calling user thread. If the main thread calls `pthread_exit`, it should join on all currently active threads, and then exit the process.

`tid_t sys_pthread_join(tid_t tid)`

Suspends the calling thread until the thread with TID `tid` finishes. Returns the TID of the thread waited on, or `TID_ERROR` if the thread could not be joined on. It is only valid to join on threads that are part of the same process and have not yet been joined on. It is valid to join on a thread that was part of the same process, but has already terminated – in such cases, the `sys_pthread_join` call should

not block. Any thread can join on any other thread (the main thread included). If a thread joins on main, it should be woken up and allowed to run after main calls `pthread_exit` but before the process is killed (see above).

The defintions of `tid_t`, `stub_fun`, and `pthread_fun` in the kernel are in `userprog/process.h`.

User-level synchronization syscalls

`bool lock_init(lock_t* lock)`

Initializes `lock`, where `lock` is a pointer to a `lock_t` in userspace. Returns true if initialization was successful. You do not have to handle the case where `lock_init` is called on the same argument twice; you can assume that the result of doing so is undefined behavior.

`bool lock_acquire(lock_t* lock)`

Acquires `lock`, blocking if necessary, where `lock` is a pointer to a `lock_t` in userspace. Returns true if the lock was successfully acquired, false if the lock was not registered with the kernel in a `lock_init` call or if the current thread already holds the lock.

`bool lock_release(lock_t* lock)`

Releases `lock`, where `lock` is a pointer to a `lock_t` in userspace. Returns true if the lock was successfully released, false if the lock was not registered with the kernel in a `lock_init` call or if the current thread does not hold the lock.

`bool sema_init(sema_t* sema, int val)`

Initializes `sema` to `val`, where `sema` is a pointer to a `sema_t` in userspace. Returns true if initialization was successful. You do not have to handle the case where `sema_init` is called on the same argument twice; you can assume that the result of doing so is undefined behavior.

`bool sema_down(sema_t* sema)`

Downs `sema`, blocking if necessary, where `sema` is a pointer to a `sema_t` in userspace. Returns true if the semaphore was successfully downed, false if the semaphore was not registered with the kernel in a `sema_init` call.

`bool sema_up(sema_t* sema)`

Ups `sema`, where `sema` is a pointer to a `sema_t` in userspace. Returns true if the `sema` was successfully upped, false if the sema was not registered with the kernel in a `sema_init` call.

Your task will be to implement these system calls in the kernel. On every synchronization system call, you are allowed to make a kernel crossing. In other words, you do not need to avoid kernel crossings like is done in the implementation of futex.

Given user-level locks and semaphores, it's possible to implement user-level condition variables entirely at user-level with locks and semaphores as primitives. Feel free to implement condition variables if you would like, but it is not required as part of the project. The implementation will look similar to the implementation of CVs in `threads/synch.c`.

Other

`tid_t get_tid(void)`

Returns the TID of the calling thread.

Modifications to process control syscalls

You will also need to update the system calls you implemented in Project User Programs to support multiple user threads. Most of the changes you'll make are short and straightforward, but substantial changes will be made to the process control syscalls. The expected behavior of process control syscalls with respect to multithreaded user programs is outlined below:

`pid_t exec(const char* file)`

When either a single-threaded or multithreaded program exec's a new process, the new process should only have a single thread of control, the main thread. New threads of control can be created in the child process with the `pthread` syscalls.

`int wait(pid_t)`

When a user thread waits on a child process, only the user thread that called wait should be suspeneded; the other threads in the parent process should be able to continue working.

`void exit(int status)`

When `exit` is called on a multithreaded program, all currently active threads in the user program should be immediately terminated: none of the user threads should be able to execute any more user-level code. Each of the backing kernel threads should release all of its resources before terminating.

As a hint and simplifying assumption, you may assume that a user thread that enters the kernel never blocks indefinitely. You are not required to make use of this assumption, but it will make implementation of this section *much* easier. As an additional hint, in `threads/interrupt.c`, you will find the function `is_trap_from_userspace`, which will return true if this interrupt represents a transition from user mode to kernel mode, to be quite helpful.

The assumption above is not true in a number of scenarios, which our test suite simply ignores. For clarity, we list a few such scenarios:

1. A user thread calls `wait` on a child process that infinite loops.
2. Two user threads deadlock with their own user-level synchronization primitives.
3. A user thread is waiting on `STDIN`, which may never arrive.

The assumption above does not apply to the case where threads are waiting on other threads in the same process through `pthread_join`. Joiners should still be woken up with the thread they joined on is killed, and joiners on the exiting thread should also be woken up.

Exit Codes

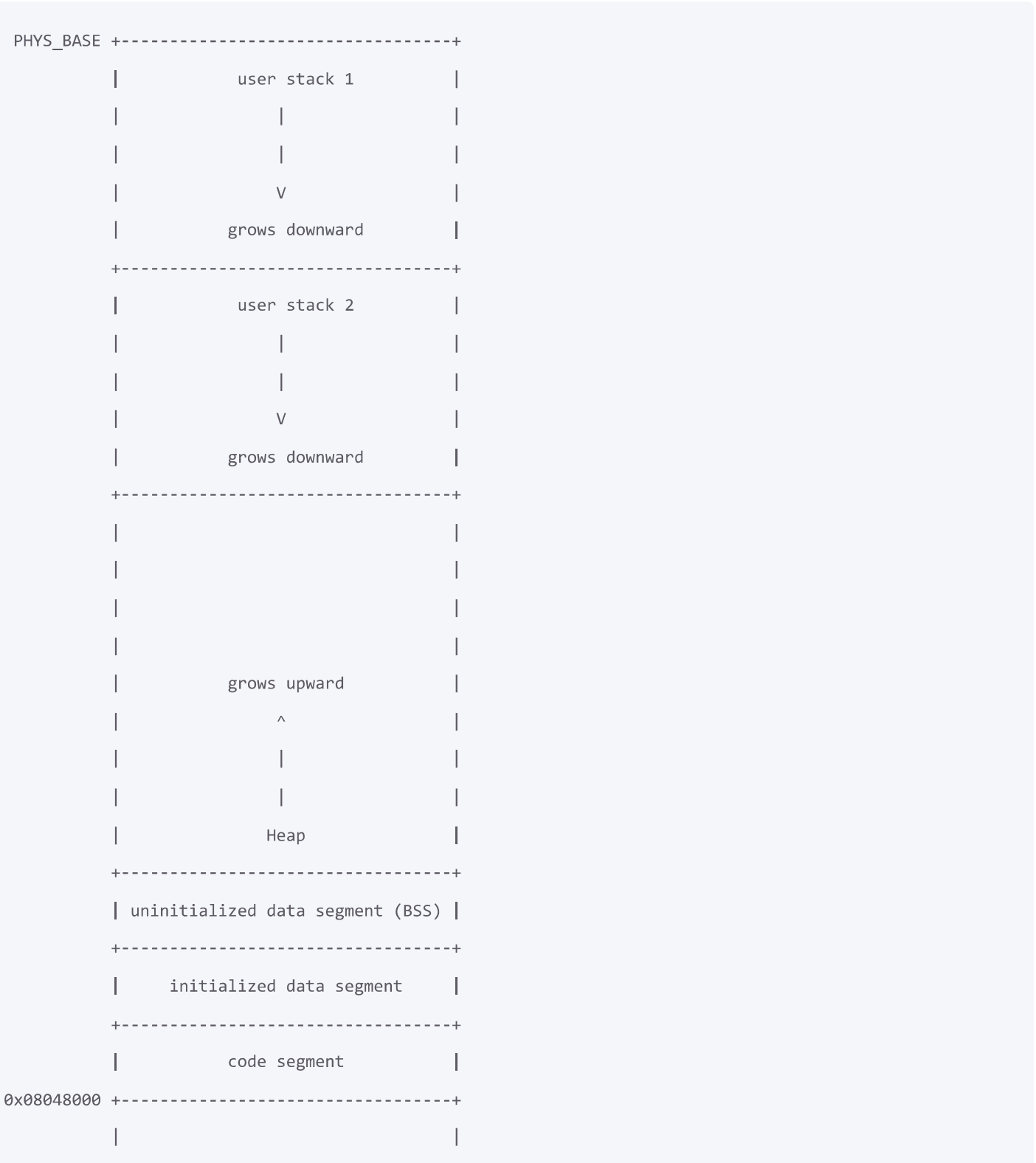
1. If the main thread calls `pthread_exit`, the process should terminate with exit code `0`.
2. If any thread calls `exit(n)`, the process should terminate with exit code `n`.
3. If the process terminates with an exception, it should exit with exit code `-1`. These are listed in priority order (with 3 being the highest priority), in the sense that if any of these occur simultaneously, the exit code should be the exit code corresponding to termination with the highest priority. For example, if main calls `pthread_exit` and while it is waiting for user threads to finish, one of them terminates with an exception, the exit code should be set to `-1`. Also, if multiple calls to `exit(n)` are made at the same time with different values of `n`, any choice of `n` is valid. Treat exit code rules as secondary: we will not test you on them in design review, and you should only be concerned about them if you are failing a test because of the wrong exit code.

Synchronization

To ease implementation difficulty, we will not be requiring you to implement fine-grained synchronization syscalls for multithreaded programs. You are allowed to serialize actions per-process (but not globally).

Implementation hints

The `setup_thread` function (found in `process.c`) should handle the creation of the user thread's stack (among other things, which are described in the docstring). In Project User Programs, since there was only one thread per process, the stack for said thread was always assigned to virtual memory addresses in the range [`PHYS_BASE - PGSIZE`, `PHYS_BASE`). In this project, there can be multiple threads per process, each of which has its own stack. Thus, you will need to choose how to assign virtual memory addresses to each thread's stack. An example diagram of an process's virtual memory where there are multiple user stacks is shown below.



```
          |                                 |
          |                                 |
          |                                 |
          |                                 |
        0 +---------------------------------+
```

Additional information

- Switching between user threads and switching between user processes require different actions on part of the kernel. Specifically, for switches between processes, the page table base pointer must be updated and any virtual caches (which for our purposes, is the TLB) should be invalidated. For switches between user threads, both of these things should be avoided. This is already done for you in `process_activate`, which is called every time a new thread is created in `load` and every time a new thread is scheduled in `thread_switch_tail`. Don't forget to activate the process when you create a new user thread.
- As our test programs are multithreaded, the `console_lock` defined in `tests/lib.c` is essential; threads can acquire this during printing calls to make sure print output of different threads is not interleaved. Currently, the test code only uses the console lock when `syn_msg` (defined in `tests/lib.c`) is set to true. The console lock is initialized in `tests/main.c` before `test_main` is called in each of the tests. Because the console lock is a user-level lock, it will only work after you have implemented user-level locking. Until you've implemented user-level locking, all your tests will fail as a result of console lock initialization; you can comment out the line `console_init()` in `tests/main.c` to temporarily prevent this issue.
