User Level Thread Library

Hingon Miu (hmiu) An Wu (anwu)
Carnegie Mellon University

1. System Stubs:

Push esi (since it's a callee-save register) before make system call, and restore later, if we need to use it.

2. Mutex:

Use an "init" variable to indicate if a mutex is initialized, so that we can check it before we do other operations. THIS IS SAME FOR COND, SEM and RWLOCK. Use an "available" variable to indicate if a mutex is available to lock. Operations on it are all ATOMIC. Use an "threads_count" variable to count the number of threads, so it won't be destroyed unless threads_count == 0.

3. Conditional Variable:

Store the waiting threads's ids in wait_tids queue so it can be FIFO.

Function implementation:

cond_wait & **cond_signal**:

cond_wait enqueue tid first, and then unlock the mutex. Use zero to deschedule (so it always succeeds).

This is race-free because in cond_signal, we also make sure that if a tid is enqueued, then we'll try to make_runnable it until success.

4. Semaphore:

Use a "count" variable to indicate the remaining available wait-without-block times for the semaphore. Also has a mutex. Use a conditional variable to deschedule current thread if sem_wait needs to block. Better than mutex because it's not busy waiting.

5. Read-write lock:

Also try to use semaphore instead of mutex to avoid busy-waiting.

Use a "read_count" to indicate the count of current readers. Also has a mutex named "mutex", because every reader needs to lock-unlock this mutex before and after they read.

Use a "rw_tids" (splay_tree) to store the thr_key (our thread id) so it's easier for us to check if a thread is reader/writer when it unlocks, and downgrade a thread. Also has a mutex named "st_mp". Use a "enter_mutex" semaphore so that every reader needs to lock-unlock it before locking (writer will acquire it for the whole process) to satisfy type-2 rwlock. (DESIGN) Use a "rw_mutex" for writer write. First-in reader will acquire it and last-out reader will release it (so writer can get in).

5. Autostack growth:

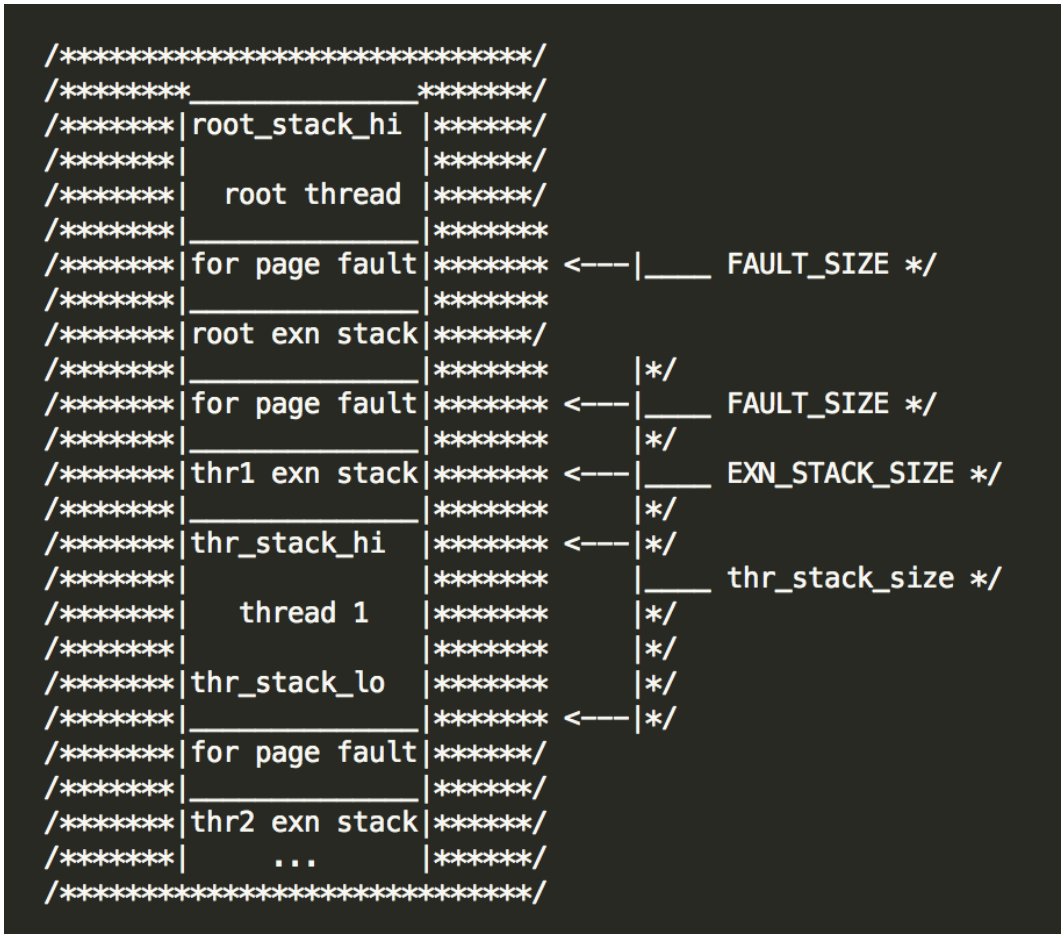
We register a autostack-growth handler on root thread so that if we detects a page fault happens, and the address of esp is lower than current stack low, we try to allocate pages up to the fault esp address, and re-register the handler.

If either the conditions don't match or we can't allocate enough esp, just don't reregister so program will page fault and be thrown to kernel (and terminate).

The exception stack of the root thread is allocated on the heap, because we don't know how long it will grow. It will be freed when multithreading starts (see next section)

6. Thread:

Multithreading stack structure:



All thread_info are stored in a splay tree structure. Every node has the real tid, the thread status, stack base, exit status, function address, arguments address, and a mutex + cv for thr_join/thr_exit.

After the first thr_create, we free the root exception stack on the heap and allocate it on the stack, 1 page after the current root stack low.

All these 1-page "fault size" is for identifying threads that get pass their stack size. We store the thr_key (our own tid) at the start of each thread's thr_stack_hi so that we can identify threads using their esp.

Function implementation:

thr_init:

Initialize all data structures and variables and store root thread's info in.

thr_create:

If it's the first time, stop root thread's autostack growth and put its exception stack on the stack (and free the heap). Allocate the child stack and store its information.

Then it does thread_fork, which takes two arguments: child_stack_base and thr_key, store thr_key on top of child_stack_base, and store child_stack_base in a caller-save register. Since all regs are guaranteed to be preserved after thread_fork, the child can switch to its stack and start running.

thr_join:

Check on a thread's status and if not EXITED, cond_wait on it. After that just free its resources.

thr_exit:

Make a thread's status EXITED and signal the possibly waiting thr_join. Before unlocking mutex, it tries to clean up and at last enqueue its stack base for reuse. After unlock just vanish so it doesn't reuse the stack anymore.