Thread Memory and Execution

Showcase Video

In our implementation, “threads” share the same struct as the proc object. They are represented by the following struct:

```
// Per-process state
struct proc {
  struct spinlock lock;
  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
    int isMain; //1 if proc/main thread, 0 if other thread

};
```

A given thread will share memory with its parent thread but have a differing trapframe, trampoline, and kstack (thus, a thread and its parent will have identical pagetable pointers except for the trapframe and trampoline pages). The main thread of a process will have isMain set to “1”, and its parent process will point to another main process. This indicates the origin of the shared memory space, initialized when procinit() is called.

All child threads of that main process have isMain set to “0” and will share memory with its parent.

Memory Sharing

Note: this feature does not work. This implementation was causing segmentation faults, so the code that makes this work in growproc() has been commented out.

When a thread requests more memory via sbrk(int n), all threads in the same family will share this allocation. The requesting thread will first receive n bytes of newly allocated pagetable entries. The thread's family (siblings and parent process if applicable)'s pagetables will then also be grown and be filled with pointers to the previously allocated pages.

Thread Creation

Threads are created via `create_thread(struct proc* thread, void* func, void* func_args)`, where the thread will begin execution at function `void* func`. A reference to the created thread will be stored in the `struct proc* thread` argument, and the function returns the newly created thread's PID.

Thread functions (func) are capable of taking one or zero uint64-sized arguments passed by `void* func_args`. To call an argumentless function, pass "0" as func_args.

Thread Joining

A thread with a given pid can be joined via system call `join_thread(int pid,uint64 addr)` . If pid is set to 0, the process will wait for whichever of its children that comes next in the proc table. The function works almost identical to xv6's wait: it halts execution of the main thread until the thread with pid "pid" finishes.

Thread Freeing

Note: this feature does not work. The method tuvmfree()
When a thread is killed, its pagetable is recursively freed via `tfreewalk(pagetable_t pagetable, int level)`, which works similar to xv6's freewalk, except it does not free actual frames (only performing the free operation when level, which indicates the level of recursion, is less than 3). This ensures that a thread's entire pagetable is freed

Thread Tests

To run a given thread test, simply type the test name into the xv6 shell.

1. Thread_funcs: Demonstrates threads' capability to start at multiple function locations and have their own stack.

2. Thread_join: Demonstrates join_thread()

3. Thread_args: Demonstrates threads' ability to receive a single uint64-sized argument

4. Mem_share: Shows how threads share memory by having them concurrently modify a global variable

5. Memleaks *(Does not work)*: Demonstrates memory security by spawning and joining 1000 threads.

6. Thread_sbrk *(Does not work)*: Demonstrates shared page allocation.

7. Reparenting tests: *Call reparent_2 immediately after reparent_1.*
- Reparent_1: Creates a parent with 5 infinitely looping threads, calls fs() to display all currently running threads, then kills the parent process.
- Reparent_2: Calls system call fs() that displays a list of all currently running threads. To show that a terminated process's unexited child threads are killed and not reparented, none of the running threads from reparent_1's initial call to fs() should be displayed.