Unix Kernel

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

1. Data Structures
(kern/data_structures/)

- **Queue**: an O(1) enqueue/dequeue queue implementation.

- **Static Queue**: Same as queue, except that the enqueue takes pre-allocated nodes.

- **Timed queue**: For sleep system call. It maintains a ticks-sorted queue so scheduler can lookup if there's any thread that wakes up in O(1) time. However the insert is O(n).

- **Circular Buffer**: use this for keyboard buffer as well as console buffer (for readline. Record user input before next line character is entered).

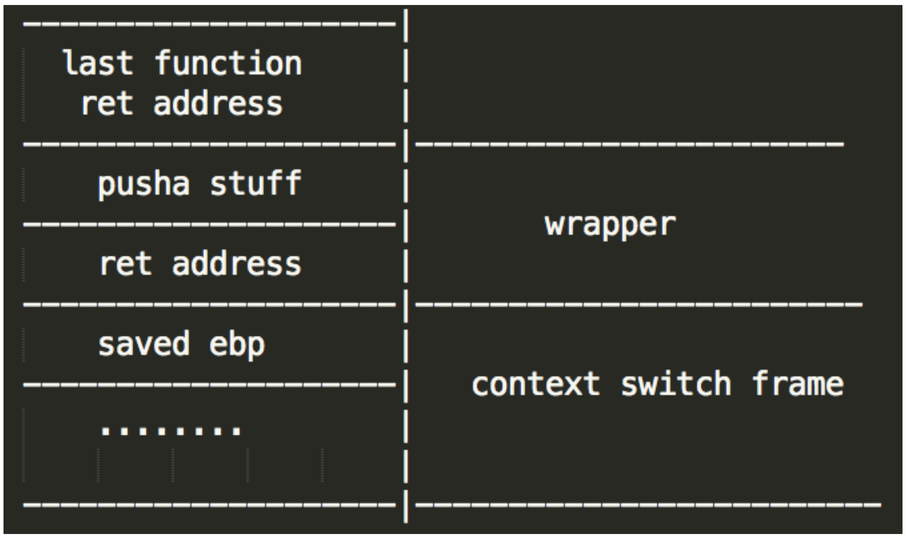
- **Hash table**: A queue array implementation. We use a hash function taken from Stack Overflow that seems to have a pretty good distribution (website link in hash table comment).

- **Static Hash table**: Same as hash table, except that the hash_insert takes pre-allocated hash entry and queue nodes.

- **Extra**: We replaced our Splay Tree structure (see user/libthread) with Hash table because of the recursive nature of tree (so it gets out of stack often).

2. Context Switch & Mode Switch
(kern/context_switch/ & kern/mode_switch.S)

For context switch, we use a wrapper of pusha/popa to save/restore register states, and we save ebp only, to restore where we should switch to. Our stack looks like:



First we execute the wrapper (which saves all registers), and then call the context switch function, which records the current ebp into kernel thread control block (ktcb), which is the address of "saved ebp" in the graph, and then do the context switch.

The context switch procedure is to load the target's ebp, move it to esp, pop ebp, and then return. After this context switch wrapper will automatically restore all the registers.

The mode switch code is very easy (2 versions). The first version, for newly loaded programs, just load all user segment registers and then iret. The second version, for swexn, will load all registers before actually mode switching.

For freshly loaded programs, in order to not have 2 paths in context switch, we set up a dummy stack for it so that it can be context switched to without exception, and will run the mode switch code right after context switch.

3. Drivers (kern/driver/)

- Timer Driver
Our timer driver is very simple: it just gets the next kernel thread that should be run (either because it wakes up from sleep, or it is the next runnable ktcb, or if the previous two do not exist, we run the first task that checks for zombies) by executing scheduler code, and context switch.

- Keyboard Driver
As requested in the handout, our keyboard driver do not context switch to the readline thread everytime a key board interrupt happens. Instead, if it checks that there's any readline thread waiting, it executes a piece of code that load the keyboard characters into a buffer, and if there's any '\n', it signals the first waiting readline thread.

4. Fault handlers (kern/exn/)

- Page fault handler
We uses Copy-On-Write implementation for fork, so for page fault it we detects that user are writing on a read-only page which is not text section or rodat section, we try to allocate a new page (or make the page writable, based on the page's reference count) and resume program execution. If that's not possible, we call the Fault handler which will be introduced below.

- Fault Handler
This handler is called when page fault handler failed to handle a case. It first checks if there are any swexn handler registered, and if so it mode switch to the handler. Otherwise, it will kill the running thread (like vanish, but set the exited status to -2).

5. Lock
(kern/lock/)

We use 2 kinds of locks in our kernel: mutex and conditional variable

- **Mutex**:
We use a non-spin-wait mutex to lock most of our resources. It has a static queue inside (so it doesn't use malloc) and if some thread locks on it while it's not available, it atomically put the thread in its queue and context switch away. When unlocking, the leaving thread will check if the queue is not empty, and if so, it will pull out a waiting thread (FIFO) and context switch to it.

- **Conditional Variable**:
The conditional variable is for readline system call and wait system call (so threads can signal them when appropriate). The cond_wait system call, in order not to be context switched away in mutex_lock (before it deschedules itself), use a special mutex_cond_lock. It's a mutex lock function that guarantees that the current thread will be switched away no matter the mutex is available or not. So in cond_wait, we first deschedule the thread in the scheduler, and call mutex_cond_lock.

6. Scheduler
(kern/sched/)

We use a round-robin scheduler for runnable threads, and a hash table for waiting threads (due to deschedule system call). The scheduler also manages a pool of running pcbs which we use to search for a process with a pid.

The scheduler's runnable pool and waiting pool are all static (node allocated on the stack) to prevent it from being context-switched when scheduling.

7. Memory
(kern/vm/)

We use Copy-On-Write for fork by recording how many times a frame is referenced by processes in a hash table.

Our page allocation functions can roll back changes if failed to alloc, in order to prevent memory leak.

However we didn't exclude frames that a process might use later when we fork the process, thus fork_bomb has page faults (because we are unable to COW, which can only be found later in our implementation).

8. Syscalls
(kern/syscall/)

- Yield
Due to our lock implementation, if someone try to yield to a specific process, and we can't find it in either runnable pool or deschedule waiting pool, it could both be exited already, or waiting on a lock. In the second case, we track down the ultimate holder (a lock holder that isn't blocking) by lock->holder->waiting_lock->holder->..., and then context switch to that thread instead of randomly yield to a random thread.

- Vanish (also Fault handler)
In vanish, if all threads of the process are exited, we free the physical frames we allocated to the process as well as its page table, and then signal the waiting parent. When a parent got a exited child by waiting, it will free child's other resources (pcb, tcb, etc.)

- Readline
Our implementation of readline rely on a queue of conditional variables: every readline system call, when enters, will create a local conditional variable and enqueue it into the cond queue (and then cond_wait). The keyboard interrupt, as introduced above, will check that if this cond queue is not empty, and if so, it will fill in the console buffer by readchar() and if there's a new line character, it will signal the first conditional variable in the queue. If a readline thread finishes reading, it will dequeue itself from the cond queue.

Because a line can be too long for one readline call, when a readline thread finishes reading and finds out that there are more available characters, it will signal the first thread in the cond queue (after itself dequeued) so that waiting readline threads can continue reading.

- Wait
The wait thread will cond_wait and wait for any of its children's signal.

9. Misc

- Preemption
We make sure that every time a new thread is available (fork/thread fork/mutex_unlock/cond_signal...), we always context switch to that process.

- Sleep/Wake
According to the hurdle, none of our threads wake up when it shouldn't.