(This is a mark down type, I also put this readme into my github. If you want to see it in better format, this is the link: https://github.com/zhangyuzhu13/MyXV6)

##Thread

In this part, I added the kernal thread into xv6 by implement two system call `clone()` and `join()`. The `clone()` is used to create a kernal thread and the `join()` is called to wait for a thread. Then I used these two system call to create a thread library with several functions so that user can use these library to use threads in their programs. Finally, I use the test case to test what I have implement and get the right result.

The main idea

I just want to use the proc structure which has already been there to represent both the thread and the process. In this way I could save much time to deal with the thread schedule. So I would add some field into the proc structure to distinguish it's a process or a thread. Also because of the way we represent the thread, the thread pid will be different even in different process. That means we could use pid to get the exact thread we want.

When implementing the clone() system call, the fields in proc structure I should change is the information about stack. There are four registers I need to know them well and change them into the proper value.

The three register is `esp`, `ebp`, `eip`, `eax`. The `esp` keeps the high address of the stack. The `ebp` keeps the low address of the stack. The stack grows from high address to the low address. The `eip` keeps the address of the function entry which will be execute by this thread. The `eax` keeps the pid.

The high address of stack should keep the pointer of argument then the return value. That means the `esp` and the `ebp` will start at the high address - 2 * sizeof(uint). The high address - sizeof(uint) keeps the argument pointer. The high address - 2 * sizeof(uint) keeps the return value which is 0xffffffff in our requirment. The `eip` should just be same with the pointer got from the parameter. The `eax` should return the its own pid if it is a thread which is different from process.

The `join()` could be similar with wait() but just deal with thread. If it's not thread then just ignore.

The key of the lock part is the atomic exchage `xchg(&lk->locked, 1)`. This atomic operation can make sure only one thread could get the lock.

There are some more details I will talk about in the rest of this part.

struct proc

First I add two more fields in the proc structure.

* int isthread
* void* stack

The first field `isthread` is used to distinguish that a proc instance is a thread or a process.

struct kthread_t

* int pid
* void* stack

We keep a pointer to stack at this struct so that we can use this pointer to free this stack in user space. In this way we can avoid the memory leak.

clone() system call

The structure of clone() function would be like this:
`int clone(void(*fcn) (void*), void *arg, void*stack)`
In the `clone()` system call, I first get the proc which is running now and copy most of the fields from the proc to the thread. What I chaneg is four register. The high address of stack should be (stack + PGSIZE - sizeof(uint)). We should put arg pointer at this address. Then put return value 0xffffffff into (stack + PGSIZE - 2 * sizeof(uint)). The `esp` and `ebp` will also initialize using the return value address. The `eax` register will keep the pid of the thread.

join() system call

Most of the `join()` part is similar with `wait()`. What different is: DO NOT FREE PAGE TABLE IF THIS IS A THREAD.

thread_create() and thread_join() function

In this function, we try to malloc a PGSIZE memory as the stack of new thread. When the address is not page aligned, we need to make the 2 PGSIZE then cut the part out of a page. Then we create a kthread_t instance which have the stack pointer and pid and return this instance.
In thread_join(), we just call join() system call then remember to free the thread stack.

Details of change

####proc.c

allocproc(): add `p->isthread = 0` which means we initialize it as process. We should change this into 1 by our selves after call this function.

wait(): add `if(p->isthread) continue` which means wait() function didnot deal with thread.

fork(): add `np->isthread = 0` which means fork will create a process not a thread.

####syscall.c

argptr(): remove `(uint)i == 0` which means the pointer could be a null pointer. It's ok if you do not dereference it but just pass it as a parameter.

Makefile

Add kthread.o to the ULIB so it's a user library and can be used. Add testkthread.c.

other files

defs.h, sysproc.c, syscall.h, user.h, usys.S. Just the files need to be changed by adding system call.

Test case

The test case create three producer and two consumer and use lock to add and delete the things to see if multithread could get the right result.

For test this part, run the xv6 OS then input:
`testkthreads`