Creating a new thread {
Just copy the contents of the fork() function and make these changes and you're done.

Replace uvmcopy() with uvmmirror(), written by you

uvmmirror: Same as uvmcopy, just drop the mem variable,
replace the reference to mem in the call to mappages with pa.
That's it.

Set np->trapframe->epc to the function pointer taken as input from the user.
what this does is that it sets the program counter to the start of that function,
so when the process is started by the scheduler, it starts from that point.

Set np->trapframe->sp to the given stack address + 4096
The reason for adding 4096 is that stack grows downwards, a push to stack reduces the value of sp. So, when the stack is empty, it should point to the very last of the page the stack resides on.
Subtract sp % 16 from sp. See line 95 of exec.c

riscv uses a0, a1, a2 ... to store function arguments (just like mips).
So, just set np->trapframe->a0 to the void* taken as input.

Set np->is_thread to 1. It'll be useful in freeproc().
}

Exiting from a thread {
Here I'm assuming that it's guaranteed that the parent thread will wait for its children to end.
Not having this assumption will only lead to more disgusting hacks.

There's literally no difference between thread_exit() and normal exit() (with the assumption).

In freeproc, there is a call to proc_freepagetable(). We'll need to modify the if statements a little bit.
If p->is_thread == 0, then just call proc_freepagetable() like before.
Otherwise, observe that proc_freepagetable() calls uvmfree() on its last line.
Inside uvmfree, there's this:

```
if(sz > 0)
    uvmunmap(pagetable, 0, PGROUNDUP(sz)/PGSIZE, 1);
freewalk(pagetable);
```

uvmunmap() just unmaps the page table, so that it's not connected to any actual memory.
The last parameter is a boolean. It controls wether the actual physical memory will be deleted.
If we are clearing a thread, we don't want to do this. This is the parent process's (recursively) job.

So, set it to 0.
}

Joining a thread {
Almost exactly same as wait(). Just add pp->pid == pid (user input) inside the if statement in the loop.
Also, drop the addr parameter and the if statement referencing it, those aren't needed.
}

If you follow upto here, your threads will run.

Note: It's the user's responsibility to put mutex locks around the calls to malloc() to make it thread safe.
Since malloc is a user program, it runs entirely on the user space,
the kernel can't do anything abount it.

Setting up proc.h {
We've already seen what to do with is_thread.
2 more variables are needed: mem_id and memlock.

Think of mem_id as the id of the physical memory the pagetable is poining to.
Threads share the same physical memory with their parent, so a parent and its children should have the same mem_id.
Allocate mem_id in exactly the same way xv6 allocates pid in allocproc().
When creating a thread, just set child->mem_id = parent->mem_id.
It will be useful in growproc().

memlock is for ensuring that two threads don't change the size of the memory at the same time,
since it might lead to race conditions. Mainly used in growproc(). Also, throw a pair of acquire and release
around any part of code in proc.c that deals with pagetables.

There are two different ways to handle memlock:
1. Let every process with the same mem_id share a single spinlock pointer.
This ensures that if a lock is held on one process, its related threads won't
be able to change anything untill the lock is released. It's just a little bit trickier
to set up, but saves a lot of headache later. I'll not go into the details of setting it up.

2. Every process has its own memlock.
You'll have to make sure that locks on all processes with the same mem_id is held inside growproc().
The reason for this is that you wouldn't want another process to increase the pagetable's size

while another process is in the process of doing it.
Easier to set up, but a lot, lot messier to use.
}

Modifying growproc() {
Acquire memlock(s) just at the start and release at the end to prevent other threads
from messing with it.

Keep everything upto the line p->sz = sz; unchanged.

After that, loop over all processes with the same mem_id (excluding self), since the change in size should be
available to all threads.
There are two conditions here: (1) n >= 0 and (2) n < 0

1. Recall what you did in uvmmirror(). There, we iterate from 0 to sz, and map them.
   But in this case, 0 to sz are already mapped, and the new part is from the old size of the process,
   to the new size (sz).
   So, make another version of uvmmirror() (uvmrangemirror() or something),
   and call mappages on PGROUNDUP(old size) to sz, instead of 0 to sz.

2. We'll need to use uvmunmap(), just like in freeproc().
   There, we started unmapping from 0,
   Here, we will start from PGROUNDUP(new size). Note that new size < old size.
   The third parameter will be (PGROUNDUP(old size) - PGROUNDUP(new size)) / PGSIZE;

PGROUNDUP is used to start from page boundaries, otherwise there will be a kernel panic.

Lastly, set the sz variable of each process to the new size.
}