Documentation Github

Thread Creation

There are two newly created user space functions that help generate and exit a thread. The first one would be **uspork_create**, which is a user available function that generates the threads.

```
int
uspork_create(void(*func)(void*), void *arg, void *stack)
{
  return clone(func, arg, stack);
}
```

Image 1: uspork_create

uspork_create is mainly just a wrapper function that calls a system call that is defined in the kernel. That system call is **clone**, which will be the main function that includes the details about thread generation. An image of **clone** can be seen below at image 2.

```
int clone(void (*func)(void*), void *arg, void *stack)
{
  //int num = *(int*) arg;
  //printf("clone has been called: %p, argument: %d, stack: %p\n", func, num, stack);

  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy user memory from parent to child.

  if(copypasta_taishi(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;

  np->parent = p;

  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);

  // set argument for the thread
  np->trapframe->a0 = (uint64)arg;

  // set return address to thread function
  np->trapframe->epc = (uint64)func;
  //np->trapframe->ra = (uint64)func;

  // set up new stack :((
  uint64 nstack = (uint64)stack;
  np->trapframe->sp = nstack;

  // flag that the thread is a thread
  np->isThread = 1;

  acquire(&runq_lock);
  list_add_tail(&runq, &np->runq_list);
  release(&runq_lock);

  // increment reference counts on open file descriptors.
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);

  safestrcpy(np->name, p->name, sizeof(p->name));

  pid = np->pid;

  release(&np->lock);

  acquire(&wait_lock);
  np->parent = p;
  release(&wait_lock);
```

Image 2: clone

The raw code can be found on the github link provided at the top of this pdf. Clone is similar to fork. While fork creates a new process, creates another memory space, and then allocates that process to that memory, clone just does part of what fork does. Since this is a lightweight thread created in xv6, there are no real things as a thread struct, but rather a process that acts like a thread. What clone does it simply allocate a new stack that the user puts in from the argument by assigning the stack pointer in the trapframe and copy the exact same memory address as the parent. This allows the process to act like a thread by living in the same memory address as the main process. The main part of clone that makes this happen would be our **copypasta_taishi** function. This function is similar to uvmcopy but instead of using memmove to create another memory space, it just copies the newly created pagetable to the old physical address. This is the core of what makes this process a thread when calling clone. Clone will eventually then return a pid specific to that thread, which will be used by join.

The second main function is **uspork_join**. This function is also a wrapper function for a system call that waits for a thread to finish. It can be seen from image 3 down below.

```
6 int
7 uspork_join(int pid, void **stack)
8 {
9   return join(pid, stack);
0 }
```

Image 3: uspork_join

The main point of join is to wait for the specific pid of the created thread created by clone. This will allow uspork join to know which thread to wait for specifically instead of just waiting for a random one. This is a more direct approach to wait(), which waits for just any thread that has finished. Join is also a blocking statement. By calling join, it will block all the other statements before it as it waits for the thread to finish running and exit. If join is not called, the thread will continue to run but no one will wait for it and the output may be jumbled with other code. Join can be seen below from image 4.

```
int join(int pid, void **stack)
{
  printf("join has been called with pid: %d, stack addr: %p\n", stack);
  struct proc *pp;
  struct proc *p = myproc();

  int pid_found = -1;
  //uint64 user_stack;

  acquire(&wait_lock);

  for(;;)
  {
    pid_found = -1;
    for (pp = proc; pp < &proc[NPROC]; pp++)
    {
      if (pp->pid == pid) {
        acquire(&pp->lock);
        if (pp->state == ZOMBIE) {
          pid_found = pp->pid;
          /*
          user_stack = pp->trapframe->sp;
          if (stack) {
            *stack = (void*)(user_stack - PGSIZE);
          }
          */
          printf("from join i gotchu gng pid: %d\n", pid_found);
          freeproc(pp);
          release(&pp->lock);
          release(&wait_lock);
          return 0;
        }
        release(&pp->lock);
      }
    }
    sleep(p, &wait_lock);
  }

  return 999;
}
```

Image 4: join

These 2 system calls work together to help achieve a shared memory. This allows the most important part, which allows all the threads created by the same process to have access to a global variable as long as there are no concurrency statements or race conditions. This will be explained a bit more in the user test cases.

Graceful Exit

Graceful exit was not handled with a specific thread_exit function, but rather through reference counting. By implementing reference counting, a repeated process that share the same memory will increase the reference count by one. By doing so, every time a thread exits, if the main process is still alive, the memory will not truly be freed through kfree. Only when the reference count reaches 0, will the process then truly be freed.

New page allocation

The new page allocation did not work correctly unfortunately. When a thread allocates a new page by calling **sbrk**, all the other threads should have access to that page. A linked list was created to link all the threads together so they have access to each other. The linked list works fine as the print statements tested when looping through the list is equal to the amount of threads being created. However, when trying to implement change on growproc or create a similar function to uvmalloc, it does not work. If there was more time, this is the problem that will be tackled first.

User Tests

The user tests are simple tests to test that our functions and theory actually works. Below are the list of tests that we have. (the first letters are lower case)

- Xv6test
    - This is the simplest test of all. Just a create and a join for some threads. This allows us to know that at least the most simple thread creation and thread joining works. This test case works perfectly.
- Sharedmem
    - By adding a global variable to increment, two different threads and also just another of the same thread should all be able to access the global variable. Since concurrency locks are not implemented in this project, sleep is used to avoid any concurrency issues. This test case works perfectly.
- Sbrktest
    - This test case makes a thread call sbrk(4096), which allocates another page when it is a thread already. The test case for this exists since the change has been done to growproc and another function was created to help with the new growproc, but unfortunately it does not work. This test case did not succeed.
- Gracefulexit
    - This test case is just an extra to show that all the threads can be joined even when waiting for a long time. This test case creates threads and lets the thread print statements. After all the thread have joined, a final print statement inside the main process will check if all of the threads have been joined and exited gracefully by checking the return values. This test case works perfectly.