OPERATING SYSTEMS LAB

XV6

Assignment : Threads

Objective

Implementing threads in xv6. Specifically,

- int thread_create(func_pointer, arg_pointer, stack_pointer),
- thread_exit(return_val_pointer) and
- thread_join(thread_id, return_val_pointer) system calls.

Background

What are threads?

A thread is a lightweight process that can :

- Operate independantly from other threads, providing concurrency.
- Shares resources with its parent process like files, but has its own stack to carry out computation.

Threads are a trade off between concurrency and indpendance, i.e. between having one big process doing all the work vs 'n' full fledged independant processes doing work in parallel.

Currently xv6 only supports creating full processes through fork() system call.

You should get yourself familiar with threads, the working of **fork()** system call and memory allocation of processes, and how a **context switch** takes place.
Furthermore, you should be aware that threads can be supported at user level or kernel level by the operating system.

Files

proc.h

You need to add tid to store thread_id, and a void * retval to store return value to the proc structure.

proc.c

In the **allocproc()** function, that when a process is created the default tid is 1. You can do this by adding

```
p->tid = 1;
```

under the “found” label.

Add the create_thread function :

```
int thread_create(void (*function)(void *), void *arg, void
*stack)
{
  int next_tid = 1, num_of_thread = 0;
  struct proc *np; // new thread
  struct proc *curproc = myproc(); // current process
  //struct proc *main_thread; // store main thread
  struct proc *p;
  int pid = curproc->pid;



  acquire (&ptable.lock);


  // find next tid
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      if (p->tid > next_tid)
      {
     next_tid = p->tid;
      }
      if (p->state != UNUSED)
      {
     num_of_thread++;
      }
    }
  }

  next_tid++;
  release(&ptable.lock);

  if (num_of_thread >= 8)
  {
    return -1; // if there is already 8 thread in the process
  }

  if((uint)stack%PGSIZE != 0)
  {
     return -1;
  }
```

```
  // allocate thread
  if((np = allocproc()) == 0){
    return -1;
  }

  np->tid = next_tid; // set tid
  np->pgdir = curproc->pgdir;
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;


  int *sp = stack + 4096 - 8;

  // moving arg to function
  np->tf->eip = (uint)function;
  np->tf->esp = (uint)stack + 4096 - 8; // top of stack
  np->tf->ebp = (uint)stack + 4096 - 8;
  np->tf->eax = 0; // initialize return value

  *(sp + 1) = (uint)arg;
  *sp = 0xffffffff;



 int i;
 for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));
  np->pid = curproc->pid;


  acquire(&ptable.lock);
  np->state = RUNNABLE;
  release(&ptable.lock);


  return next_tid;
}
```

Some points regarding the thread_create function :

- It mimics the fork() system call, except assigning shared resources to make it lightweight and some other changes.
- It supports maximum of 8 threads per process.
- It returns the assigned tid.

- Each thread has its own stack. We assume that this is supplied by the user. This can easily be included in the kernel code itself.

Explanation

The following code gets highest tid and number of threads.

```
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      if (p->tid > next_tid)
      {
     next_tid = p->tid;
      }
      if (p->state != UNUSED)
      {
     num_of_thread++;
      }
    }
  }
  next_tid++;
  release(&ptable.lock);

  if (num_of_thread >= 8)
  {
    return -1;
  }
```

The stack must be page aligned (each pageis 4096 words). We also check that the stack is page aligned or not.

```
if((uint)stack%PGSIZE != 0)
  {
     return -1;
  }
```

Allocate a new process.

```
if((np = allocproc()) == 0){
    return -1;
  }
```

First we initialize the thread with values that it shares with the parent process. These are page table i.e. memory, memory size, and its parent.

```
  np->pgdir = curproc->pgdir;
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;
```

Now we need to initialize the function that will run as thread. This is done by setting values of registers that will be used during/after context switch.

- eip holds the next instruction to be executed.
- esp points to top of stack.
- eax holds the return value, this is also used in fork to return 0 as pid to the child.

```
 np->tf->eip = (uint)function;
  np->tf->esp = (uint)stack + PGSIZE - 8; // top of stack
  np->tf->ebp = (uint)stack + PGSIZE - 8;
  np->tf->eax = 0;
```

Putting arguments where a function wil look for them, and setting a fake return value.

```
*(sp + 1) = (uint)arg;
*sp = 0xffffffff;
```

Copying files, process name, process id from parent.

```
int i;
 for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));
// copy process name
  np->pid = curproc->pid;
```

Exiting a thread

- This will be called from inside the thread when it wants to finish.
- Passes the return value.

The code :

```
void thread_exit(void *retval)
{
  int fd, pid, tid;
  struct proc *curproc = myproc();
  struct proc *p;

  pid = curproc->pid;
  tid = curproc->tid;
```

```
  if (curproc == initproc)
    panic("init exiting");


  if (tid == 1) // terminate all thread
  {
    exit();
  }

  else // terminate thread only who has that tid
  {
    for(fd = 0; fd < NOFILE; fd++){
      if(curproc->ofile[fd]){
        fileclose(curproc->ofile[fd]);
        curproc->ofile[fd] = 0;
      }
    }

    begin_op();
    iput(curproc->cwd);
    end_op();
    curproc->cwd = 0;
    acquire(&ptable.lock);
    wakeup1(curproc->parent);

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (p->parent == curproc) {
        p->parent = initproc;
        if (p->state == ZOMBIE)
       wakeup1(initproc);
      }
    }

    // wake up sleeping threads
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->pid == pid && p->tid != tid)
     wakeup1(p);
    }
    //

    curproc->state = ZOMBIE;
    curproc->retval = retval;

    sched();
    panic("zombie exit");
  }
}
```

Joining a thread

- Called by the parent, to wait for its child thread to exit.
- Also frees up resources and resets values.
- Collects retval value stored in the thread's data structure.

Code :

```
int thread_join(int tid, void **retval)
{
  struct proc *p;
  int find_threads;
  struct proc *curproc = myproc();
  int pid = curproc->pid;
  find_threads = 0;

  acquire(&ptable.lock);
  for(;;)
  {
    // Scan through table looking for exited children.
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if(p->pid != pid)
     continue;

      if (p->tid == tid)
      {
        find_threads = 1;
     if(p->state == ZOMBIE)
        {
          // Found one.
          kfree(p->kstack);
          p->parent = 0;
          p->name[0] = 0;
          p->killed = 0;
          p->pid = 0;
          p->tid = 0;
          p->state = UNUSED;
          *retval = (void *)p->retval;
          p->retval = 0;
          release(&ptable.lock);
          return 0;
     }

        // if thread is already terminated
        if(p->state == UNUSED)
        {
          release(&ptable.lock);
          return 0;
        }
      }
    }
```

```
    // No point waiting if we don't have any children.
    if(curproc->killed)
    {
      release(&ptable.lock);
      return -1;
    }

    // if there is no thread who has that tid
    if (find_threads == 0)
    {
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in
proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
  return 0;
}
```

Also add the following utility function :

```
int gettid(void)
{
  struct proc *curproc = myproc(); // current process
  return curproc->tid;

}
```

defs.h

Add the following to facillitate system calls.

- int thread_create(void (*function)(void *), void *arg, void *stack);
- void thread_exit(void *retval);
- int thread_join(int tid, void **retval);
- int gettid(void);

user.h

- int thread_create(void (*function)(void *), void *arg, void *stack);
- void thread_exit(void *retval);
- int thread_join (int tid, void **retval);
- int gettid(void);

usys.S, syscall.h, syscall.c

Add definitions to implement the system calls as in previous labs.

sysproc.c

Argptr and **argint** are functions used to fetched arguments.

```
int
sys_thread_create(void)
{
  void (* fcn)(void *);
  void *arg, *stack;
  int ret;
  ret = -1;
  if (argptr(0, (void *)&fcn, sizeof(fcn) < 0)) return ret;
  if (argptr(1, (void *)&arg, sizeof(arg) < 0)) return ret;
  if (argptr(2, (void *)&stack, sizeof(stack) < 0)) return
ret;
  ret = thread_create((void *)(fcn), (void *)arg, (void
*)stack);
  return ret;
}
```

```
int
sys_thread_exit(void)
{
  void *retval;
  if (argptr(0, (void *)&retval, sizeof(retval) < 0)) return
-1;

  thread_exit(retval);
  return 1;
}

int
sys_thread_join(void)
{
  int tid;
  void **retval;
  if (argptr(0, (void *)&tid, sizeof(tid) < 0)) return -1;
  if (argint(1, (int*)&retval) < 0) return -1;
  return thread_join(tid, retval);
}

int
sys_gettid(void)
{
  return gettid();
}
```

After implementing the system call, you should run the **threadtest.c** file that has been provided with this assignment, to check the functioning of your threads.

References :

https://web.stanford.edu/~ouster/cgi-bin/cs140-spring14/lecture.php?topic=thread

https://www.ics.uci.edu/~aburtsev/238P/2018winter/hw/hw4-threads.html

https://github.com/contestpark/xv6-public/tree/master/p2

https://download-mirror.savannah.gnu.org/releases/pgubook/ProgrammingGroundUp-1-0-booksize.pdf