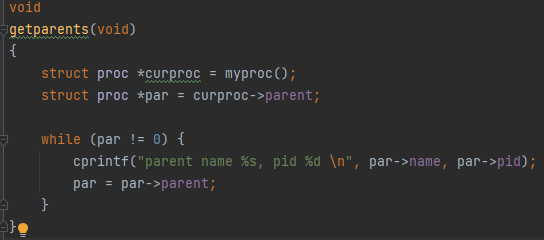
**Lab 1 Report – Ethan Phuong**

**Part 1:**

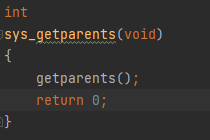
In this section, we were tasked with adding a new system call getparents(). For this change, I had to change the proc.c, defs.h, sysproc.h, syscall.h, syscall.c, user.h, and usys.s files. The following image are my changes in proc.c where I added the getparents() function. I accessed the current process’s parent and then printed out the current process’s parent until I reached the initial process whose parent is null. I updated the parent each time so I could get the parent’s parent every time.



Below is defs.h where I added the function to the proc.c section of the file.



Below is the system call I added to sysproc.c. Getparents() is a void program that is supposed to print so I only called it in the system call.



Below is the addition I made for getparents in syscall.h.



Below are the additions to syscall.c.





Below is the addition to user.h.

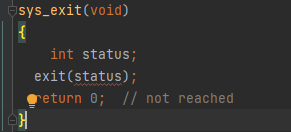


Below is the addition to usys.s.



**Part 2:**

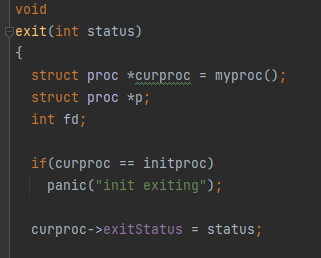
In this section, we were tasked with extending the xv6 process implementation to include an exit status. This portion was in part preparation for part 3. In this section, I had to change the proc.h file and also changed the sysproc.c file for the time being.



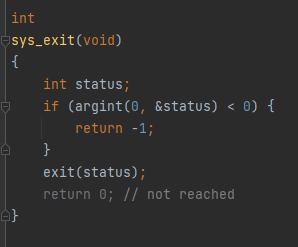
On the left is the exitStatus variable that was necessary to implement exit status. Looking forward to part 3, I created the variable in type int because that is the parameter that is being taken in for the new implementation of exit. This change was done in the proc.h file. On the right is the change I made in sysproc.c. I added a status variable and passed it through the exit call in preparation for the next part.

**Part 3:**

In this section, we were tasked with changing the call signature for exit to exit(int status). This required changes in many files because of the implications of changing exit(). In proc.c, I added the curproc->exitStatus = status so that I could maintain the current status when the process is terminated. Below is the change I made in proc.c



Below is the change I made in sysproc.c. I added the if condition to take in the status argument as an int.



Below I changed the parameter in defs.h to int to fit the function.



I also changed it to int in user.h.

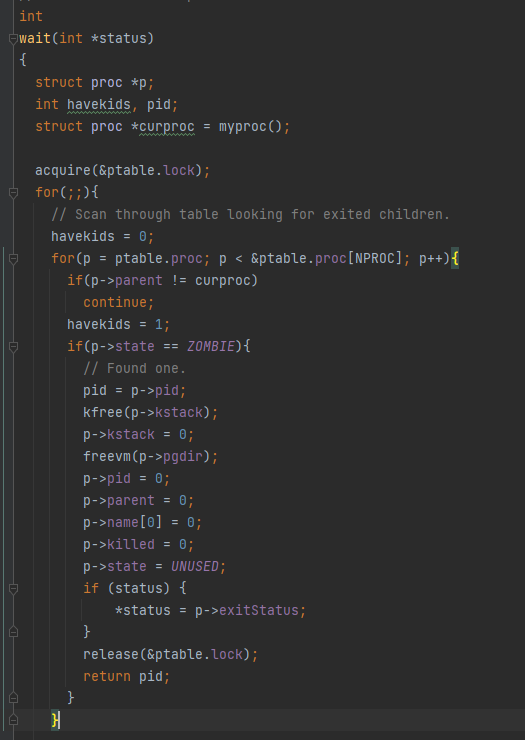


In zombie.c, wc.c, usertests.c, trap.c, stressfs.c, sh.c, rm.c, mkdir.c, ls.c, ln.c, cat.c and many others I changed exit(); to exit(0); because the function now takes in an int as a parameter.

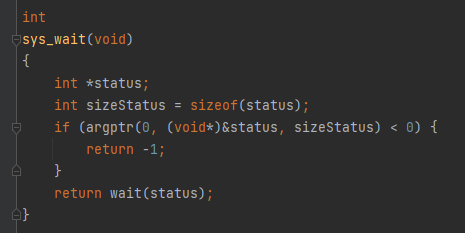


**Part 4:**

In this part we had to update wait to wait(int \*status). To complete this I had to update proc.c, sysproc.c, user.h, and defs.h. Proc.c was the main change where it must prevent termination and then pass the exit status to the status variable. While the function is going through the ptable we have to include the condition so that it does not return. Below is the change I made in proc.c. I added an if statement that checks the status and then saves the status as the exit status.



Below are the changes I made to the system call. Similar to exit, I added the conditional to take the status argument, this time changing it to take status as a pointer.



Below, I changed the parameter of wait to int\* in user.h



I also changed the parameter of wait to int\* in defs.h

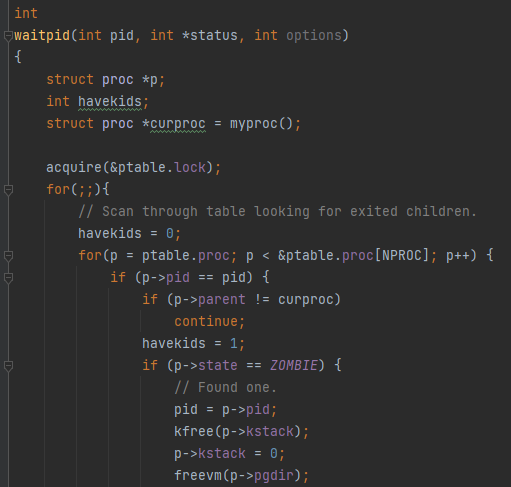


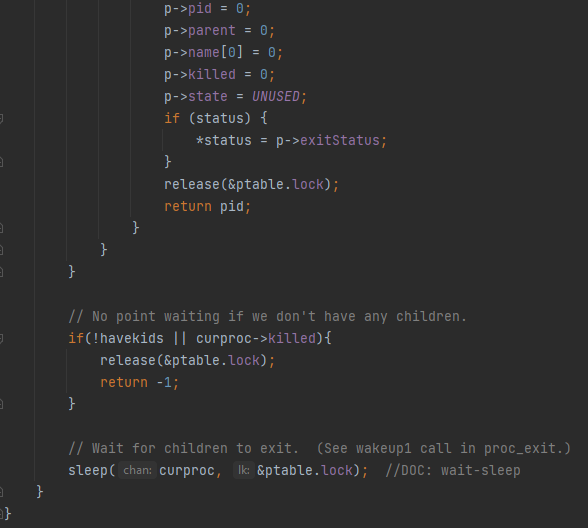
**Part 5:**

In part 5 we added a new system call called waitpid. This function was required to wait for a similar process pid to the one provided in the argument. The function takes in a pid, a status, and an options that is not used right now. Below is the change I made to proc.c to check if the pids are equivalent.

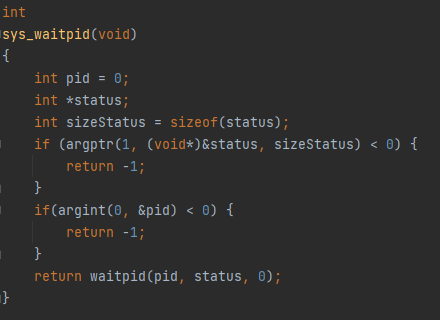


Besides that line, the function is alike to wait and the rest of the function is copied from wait(). Below is the entire waitpid function in proc.c.





Below are the changes that I made to sysproc.c. To take the pid and status I added the two conditionals, where the status pointer is the argument in the 1 spot and the int pid is the argument in the 0 spot.



Below is the definition added in syscall.h



Below are the definitions for the system call added in syscall.c





Below is the definition added in usys.s



Below is the definition added in defs.h

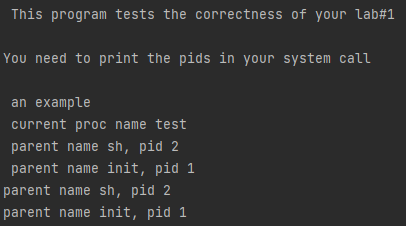


Below is the definition added in user.h

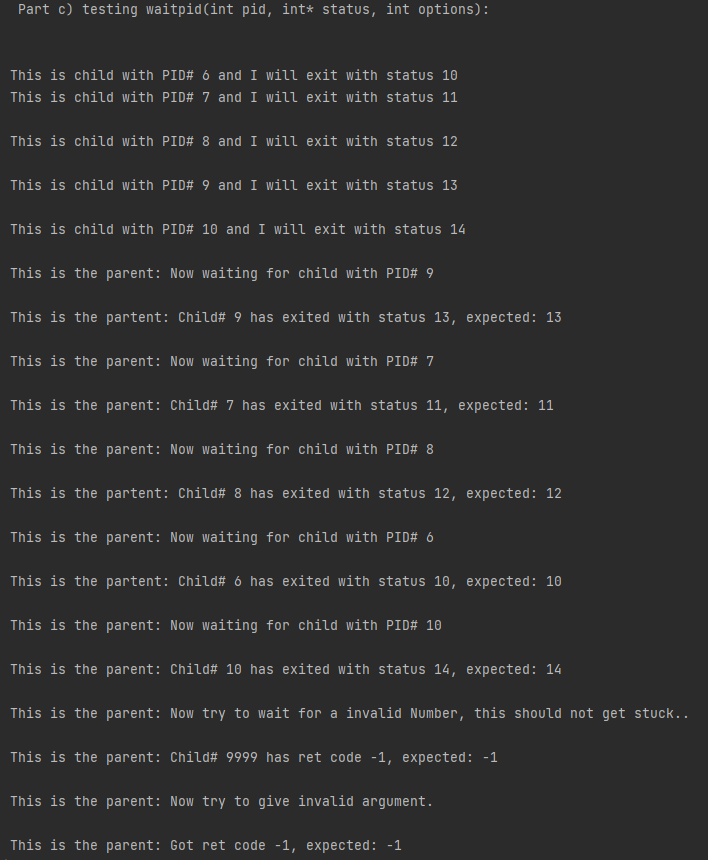


**Results:**

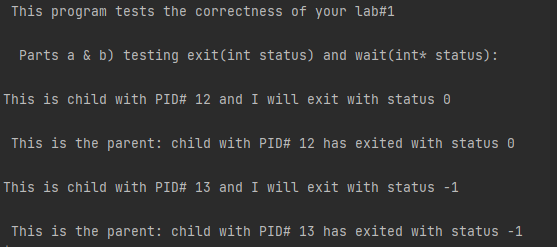
**Test 1:**

****

**Test 2:**

****

**Test 3:**

****

The tests used for all these results were from the test.c file provided in the slides.

**difffile.txt:**

diff --git a/Lab 1/xv6/proc.c b/Lab 1/xv6/proc.c

index e7662f5..10fe4d9 100644

--- a/Lab 1/xv6/proc.c

+++ b/Lab 1/xv6/proc.c

@@ -1,599 +1,599 @@

-#include "types.h"

-#include "defs.h"

-#include "param.h"

-#include "memlayout.h"

-#include "mmu.h"

-#include "x86.h"

-#include "proc.h"

-#include "spinlock.h"

-

-struct {

- struct spinlock lock;

- struct proc proc[NPROC];

-} ptable;

-

-static struct proc \*initproc;

-

-int nextpid = 1;

-extern void forkret(void);

-extern void trapret(void);

-

-static void wakeup1(void \*chan);

-

-void

-pinit(void)

-{

- initlock(&ptable.lock, "ptable");

-}

-

-// Must be called with interrupts disabled

-int

-cpuid() {

- return mycpu()-cpus;

-}

-

-// Must be called with interrupts disabled to avoid the caller being

-// rescheduled between reading lapicid and running through the loop.

-struct cpu\*

-mycpu(void)

-{

- int apicid, i;

-

- if(readeflags()&FL\_IF)

- panic("mycpu called with interrupts enabled\n");

-

- apicid = lapicid();

- // APIC IDs are not guaranteed to be contiguous. Maybe we should have

- // a reverse map, or reserve a register to store &cpus[i].

- for (i = 0; i < ncpu; ++i) {

- if (cpus[i].apicid == apicid)

- return &cpus[i];

- }

- panic("unknown apicid\n");

-}

-

-// Disable interrupts so that we are not rescheduled

-// while reading proc from the cpu structure

-struct proc\*

-myproc(void) {

- struct cpu \*c;

- struct proc \*p;

- pushcli();

- c = mycpu();

- p = c->proc;

- popcli();

- return p;

-}

-

-//PAGEBREAK: 32

-// Look in the process table for an UNUSED proc.

-// If found, change state to EMBRYO and initialize

-// state required to run in the kernel.

-// Otherwise return 0.

-static struct proc\*

-allocproc(void)

-{

- struct proc \*p;

- char \*sp;

-

- acquire(&ptable.lock);

-

- for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)

- if(p->state == UNUSED)

- goto found;

-

- release(&ptable.lock);

- return 0;

-

-found:

- p->state = EMBRYO;

- p->pid = nextpid++;

-

- release(&ptable.lock);

-

- // Allocate kernel stack.

- if((p->kstack = kalloc()) == 0){

- p->state = UNUSED;

- return 0;

- }

- sp = p->kstack + KSTACKSIZE;

-

- // Leave room for trap frame.

- sp -= sizeof \*p->tf;

- p->tf = (struct trapframe\*)sp;

-

- // Set up new context to start executing at forkret,

- // which returns to trapret.

- sp -= 4;

- \*(uint\*)sp = (uint)trapret;

-

- sp -= sizeof \*p->context;

- p->context = (struct context\*)sp;

- memset(p->context, 0, sizeof \*p->context);

- p->context->eip = (uint)forkret;

-

- return p;

-}

-

-//PAGEBREAK: 32

-// Set up first user process.

-void

-userinit(void)

-{

- struct proc \*p;

- extern char \_binary\_initcode\_start[], \_binary\_initcode\_size[];

-

- p = allocproc();

-

- initproc = p;

- if((p->pgdir = setupkvm()) == 0)

- panic("userinit: out of memory?");

- inituvm(p->pgdir, \_binary\_initcode\_start, (int)\_binary\_initcode\_size);

- p->sz = PGSIZE;

- memset(p->tf, 0, sizeof(\*p->tf));

- p->tf->cs = (SEG\_UCODE << 3) | DPL\_USER;

- p->tf->ds = (SEG\_UDATA << 3) | DPL\_USER;

- p->tf->es = p->tf->ds;

- p->tf->ss = p->tf->ds;

- p->tf->eflags = FL\_IF;

- p->tf->esp = PGSIZE;

- p->tf->eip = 0; // beginning of initcode.S

-

- safestrcpy(p->name, "initcode", sizeof(p->name));

- p->cwd = namei("/");

-

- // this assignment to p->state lets other cores

- // run this process. the acquire forces the above

- // writes to be visible, and the lock is also needed

- // because the assignment might not be atomic.

- acquire(&ptable.lock);

-

- p->state = RUNNABLE;

-

- release(&ptable.lock);

-}

-

-// Grow current process's memory by n bytes.

-// Return 0 on success, -1 on failure.

-int

-growproc(int n)

-{

- uint sz;

- struct proc \*curproc = myproc();

-

- sz = curproc->sz;

- if(n > 0){

- if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)

- return -1;

- } else if(n < 0){

- if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)

- return -1;

- }

- curproc->sz = sz;

- switchuvm(curproc);

- return 0;

-}

-

-// Create a new process copying p as the parent.

-// Sets up stack to return as if from system call.

-// Caller must set state of returned proc to RUNNABLE.

-int

-fork(void)

-{

- int i, pid;

- struct proc \*np;

- struct proc \*curproc = myproc();

-

- // Allocate process.

- if((np = allocproc()) == 0){

- return -1;

- }

-

- // Copy process state from proc.

- if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){

- kfree(np->kstack);

- np->kstack = 0;

- np->state = UNUSED;

- return -1;

- }

- np->sz = curproc->sz;

- np->parent = curproc;

- \*np->tf = \*curproc->tf;

-

- // Clear %eax so that fork returns 0 in the child.

- np->tf->eax = 0;

-

- 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));

-

- pid = np->pid;

-

- acquire(&ptable.lock);

-

- np->state = RUNNABLE;

-

- release(&ptable.lock);

-

- return pid;

-}

-

-// Exit the current process. Does not return.

-// An exited process remains in the zombie state

-// until its parent calls wait() to find out it exited.

-void

-exit(int status)

-{

- struct proc \*curproc = myproc();

- struct proc \*p;

- int fd;

-

- if(curproc == initproc)

- panic("init exiting");

-

- curproc->exitStatus = status;

-

- // Close all open files.

- for(fd = 0; fd < NOFILE; fd++){

- if(curproc->ofile[fd]){

- fileclose(curproc->ofile[fd]);

- curproc->ofile[fd] = 0;

+ #include "types.h"

+ #include "defs.h"

+ #include "param.h"

+ #include "memlayout.h"

+ #include "mmu.h"

+ #include "x86.h"

+ #include "proc.h"

+ #include "spinlock.h"

+

+ struct {

+ struct spinlock lock;

+ struct proc proc[NPROC];

+ } ptable;

+

+ static struct proc \*initproc;

+

+ int nextpid = 1;

+ extern void forkret(void);

+ extern void trapret(void);

+

+ static void wakeup1(void \*chan);

+

+ void

+ pinit(void)

+ {

+ initlock(&ptable.lock, "ptable");

}

- }

- begin\_op();

- iput(curproc->cwd);

- end\_op();

- curproc->cwd = 0;

-

- acquire(&ptable.lock);

+ // Must be called with interrupts disabled

+ int

+ cpuid() {

+ return mycpu()-cpus;

+ }

- // Parent might be sleeping in wait().

- wakeup1(curproc->parent);

+ // Must be called with interrupts disabled to avoid the caller being

+ // rescheduled between reading lapicid and running through the loop.

+ struct cpu\*

+ mycpu(void)

+ {

+ int apicid, i;

+

+ if(readeflags()&FL\_IF)

+ panic("mycpu called with interrupts enabled\n");

+

+ apicid = lapicid();

+ // APIC IDs are not guaranteed to be contiguous. Maybe we should have

+ // a reverse map, or reserve a register to store &cpus[i].

+ for (i = 0; i < ncpu; ++i) {

+ if (cpus[i].apicid == apicid)

+ return &cpus[i];

+ }

+ panic("unknown apicid\n");

+ }

- // Pass abandoned children to init.

- for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

- if(p->parent == curproc){

- p->parent = initproc;

- if(p->state == ZOMBIE)

- wakeup1(initproc);

+ // Disable interrupts so that we are not rescheduled

+ // while reading proc from the cpu structure

+ struct proc\*

+ myproc(void) {

+ struct cpu \*c;

+ struct proc \*p;

+ pushcli();

+ c = mycpu();

+ p = c->proc;

+ popcli();

+ return p;

}

- }

-

- // Jump into the scheduler, never to return.

- curproc->state = ZOMBIE;

- sched();

- panic("zombie exit");

-}

-

-// Wait for a child process to exit and return its pid.

-// Return -1 if this process has no children.

-int

-wait(int \*status)

-{

- struct proc \*p;

- int havekids, pid;

- struct proc \*curproc = myproc();

-

- acquire(&ptable.lock);

- for(;;){

- // Scan through table looking for exited children.

- havekids = 0;

- for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

- if(p->parent != curproc)

- continue;

- havekids = 1;

- if(p->state == ZOMBIE){

- // Found one.

- pid = p->pid;

- kfree(p->kstack);

- p->kstack = 0;

- freevm(p->pgdir);

- p->pid = 0;

- p->parent = 0;

- p->name[0] = 0;

- p->killed = 0;

+

+ //PAGEBREAK: 32

+ // Look in the process table for an UNUSED proc.

+ // If found, change state to EMBRYO and initialize

+ // state required to run in the kernel.

+ // Otherwise return 0.

+ static struct proc\*

+ allocproc(void)

+ {

+ struct proc \*p;

+ char \*sp;

+

+ acquire(&ptable.lock);

+

+ for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)

+ if(p->state == UNUSED)

+ goto found;

+

+ release(&ptable.lock);

+ return 0;

+

+ found:

+ p->state = EMBRYO;

+ p->pid = nextpid++;

+

+ release(&ptable.lock);

+

+ // Allocate kernel stack.

+ if((p->kstack = kalloc()) == 0){

p->state = UNUSED;

- if (status) {

- \*status = p->exitStatus;

- }

- release(&ptable.lock);

- return pid;

+ return 0;

}

+ sp = p->kstack + KSTACKSIZE;

+

+ // Leave room for trap frame.

+ sp -= sizeof \*p->tf;

+ p->tf = (struct trapframe\*)sp;

+

+ // Set up new context to start executing at forkret,

+ // which returns to trapret.

+ sp -= 4;

+ \*(uint\*)sp = (uint)trapret;

+

+ sp -= sizeof \*p->context;

+ p->context = (struct context\*)sp;

+ memset(p->context, 0, sizeof \*p->context);

+ p->context->eip = (uint)forkret;

+

+ return p;

}

- // No point waiting if we don't have any children.

- if(!havekids || curproc->killed){

+ //PAGEBREAK: 32

+ // Set up first user process.

+ void

+ userinit(void)

+ {

+ struct proc \*p;

+ extern char \_binary\_initcode\_start[], \_binary\_initcode\_size[];

+

+ p = allocproc();

+

+ initproc = p;

+ if((p->pgdir = setupkvm()) == 0)

+ panic("userinit: out of memory?");

+ inituvm(p->pgdir, \_binary\_initcode\_start, (int)\_binary\_initcode\_size);

+ p->sz = PGSIZE;

+ memset(p->tf, 0, sizeof(\*p->tf));

+ p->tf->cs = (SEG\_UCODE << 3) | DPL\_USER;

+ p->tf->ds = (SEG\_UDATA << 3) | DPL\_USER;

+ p->tf->es = p->tf->ds;

+ p->tf->ss = p->tf->ds;

+ p->tf->eflags = FL\_IF;

+ p->tf->esp = PGSIZE;

+ p->tf->eip = 0; // beginning of initcode.S

+

+ safestrcpy(p->name, "initcode", sizeof(p->name));

+ p->cwd = namei("/");

+

+ // this assignment to p->state lets other cores

+ // run this process. the acquire forces the above

+ // writes to be visible, and the lock is also needed

+ // because the assignment might not be atomic.

+ acquire(&ptable.lock);

+

+ p->state = RUNNABLE;

+

release(&ptable.lock);

- return -1;

}

- // Wait for children to exit. (See wakeup1 call in proc\_exit.)

- sleep(curproc, &ptable.lock); //DOC: wait-sleep

- }

-}

+ // Grow current process's memory by n bytes.

+ // Return 0 on success, -1 on failure.

+ int

+ growproc(int n)

+ {

+ uint sz;

+ struct proc \*curproc = myproc();

+

+ sz = curproc->sz;

+ if(n > 0){

+ if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)

+ return -1;

+ } else if(n < 0){

+ if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)

+ return -1;

+ }

+ curproc->sz = sz;

+ switchuvm(curproc);

+ return 0;

+ }

+

+ // Create a new process copying p as the parent.

+ // Sets up stack to return as if from system call.

+ // Caller must set state of returned proc to RUNNABLE.

+ int

+ fork(void)

+ {

+ int i, pid;

+ struct proc \*np;

+ struct proc \*curproc = myproc();

+

+ // Allocate process.

+ if((np = allocproc()) == 0){

+ return -1;

+ }

+

+ // Copy process state from proc.

+ if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){

+ kfree(np->kstack);

+ np->kstack = 0;

+ np->state = UNUSED;

+ return -1;

+ }

+ np->sz = curproc->sz;

+ np->parent = curproc;

+ \*np->tf = \*curproc->tf;

+

+ // Clear %eax so that fork returns 0 in the child.

+ np->tf->eax = 0;

-int

-waitpid(int pid, int \*status, int options)

-{

- struct proc \*p;

- int havekids;

- struct proc \*curproc = myproc();

+ for(i = 0; i < NOFILE; i++)

+ if(curproc->ofile[i])

+ np->ofile[i] = filedup(curproc->ofile[i]);

+ np->cwd = idup(curproc->cwd);

- acquire(&ptable.lock);

- for(;;){

+ safestrcpy(np->name, curproc->name, sizeof(curproc->name));

+

+ pid = np->pid;

+

+ acquire(&ptable.lock);

+

+ np->state = RUNNABLE;

+

+ release(&ptable.lock);

+

+ return pid;

+ }

+

+ // Exit the current process. Does not return.

+ // An exited process remains in the zombie state

+ // until its parent calls wait() to find out it exited.

+ void

+ exit(int status)

+ {

+ struct proc \*curproc = myproc();

+ struct proc \*p;

+ int fd;

+

+ if(curproc == initproc)

+ panic("init exiting");

+

+ curproc->exitStatus = status;

+

+ // Close all open files.

+ 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);

+

+ // Parent might be sleeping in wait().

+ wakeup1(curproc->parent);

+

+ // Pass abandoned children to init.

+ for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

+ if(p->parent == curproc){

+ p->parent = initproc;

+ if(p->state == ZOMBIE)

+ wakeup1(initproc);

+ }

+ }

+

+ // Jump into the scheduler, never to return.

+ curproc->state = ZOMBIE;

+ sched();

+ panic("zombie exit");

+ }

+

+ // Wait for a child process to exit and return its pid.

+ // Return -1 if this process has no children.

+ int

+ wait(int \*status)

+ {

+ struct proc \*p;

+ int havekids, pid;

+ struct proc \*curproc = myproc();

+

+ acquire(&ptable.lock);

+ for(;;){

// Scan through table looking for exited children.

havekids = 0;

- for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {

- if (p->pid == pid) {

- if (p->parent != curproc)

- continue;

- havekids = 1;

- if (p->state == ZOMBIE) {

- // Found one.

- pid = p->pid;

- kfree(p->kstack);

- p->kstack = 0;

- freevm(p->pgdir);

- p->pid = 0;

- p->parent = 0;

- p->name[0] = 0;

- p->killed = 0;

- p->state = UNUSED;

- if (status) {

- \*status = p->exitStatus;

- }

- release(&ptable.lock);

- return pid;

- }

+ for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

+ if(p->parent != curproc)

+ continue;

+ havekids = 1;

+ if(p->state == ZOMBIE){

+ // Found one.

+ pid = p->pid;

+ kfree(p->kstack);

+ p->kstack = 0;

+ freevm(p->pgdir);

+ p->pid = 0;

+ p->parent = 0;

+ p->name[0] = 0;

+ p->killed = 0;

+ p->state = UNUSED;

+ if (status) {

+ \*status = p->exitStatus;

}

+ release(&ptable.lock);

+ return pid;

+ }

}

// No point waiting if we don't have any children.

if(!havekids || curproc->killed){

- release(&ptable.lock);

- return -1;

+ release(&ptable.lock);

+ return -1;

}

// Wait for children to exit. (See wakeup1 call in proc\_exit.)

sleep(curproc, &ptable.lock); //DOC: wait-sleep

+ }

+ }

+

+ int

+ waitpid(int pid, int \*status, int options)

+ {

+ struct proc \*p;

+ int havekids;

+ struct proc \*curproc = myproc();

+

+ acquire(&ptable.lock);

+ for(;;){

+ // Scan through table looking for exited children.

+ havekids = 0;

+ for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {

+ if (p->pid == pid) {

+ if (p->parent != curproc)

+ continue;

+ havekids = 1;

+ if (p->state == ZOMBIE) {

+ // Found one.

+ pid = p->pid;

+ kfree(p->kstack);

+ p->kstack = 0;

+ freevm(p->pgdir);

+ p->pid = 0;

+ p->parent = 0;

+ p->name[0] = 0;

+ p->killed = 0;

+ p->state = UNUSED;

+ if (status) {

+ \*status = p->exitStatus;

+ }

+ release(&ptable.lock);

+ return pid;

+ }

+ }

+ }

+

+ // No point waiting if we don't have any children.

+ if(!havekids || curproc->killed){

+ release(&ptable.lock);

+ return -1;

+ }

+

+ // Wait for children to exit. (See wakeup1 call in proc\_exit.)

+ sleep(curproc, &ptable.lock); //DOC: wait-sleep

+ }

}

-}

-void

-getparents(void)

-{

- struct proc \*curproc = myproc();

- struct proc \*par = curproc->parent;

+ void

+ getparents(void)

+ {

+ struct proc \*curproc = myproc();

+ struct proc \*par = curproc->parent;

- while (par != 0) {

- cprintf("parent name %s, pid %d \n", par->name, par->pid);

- par = par->parent;

+ while (par != 0) {

+ cprintf("parent name %s, pid %d \n", par->name, par->pid);

+ par = par->parent;

+ }

}

-}

-

-//PAGEBREAK: 42

-// Per-CPU process scheduler.

-// Each CPU calls scheduler() after setting itself up.

-// Scheduler never returns. It loops, doing:

-// - choose a process to run

-// - swtch to start running that process

-// - eventually that process transfers control

-// via swtch back to the scheduler.

-void

-scheduler(void)

-{

- struct proc \*p;

- struct cpu \*c = mycpu();

- c->proc = 0;

-

- for(;;){

- // Enable interrupts on this processor.

- sti();

-

- // Loop over process table looking for process to run.

- acquire(&ptable.lock);

- for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

- if(p->state != RUNNABLE)

- continue;

-

- // Switch to chosen process. It is the process's job

- // to release ptable.lock and then reacquire it

- // before jumping back to us.

- c->proc = p;

- switchuvm(p);

- p->state = RUNNING;

-

- swtch(&(c->scheduler), p->context);

- switchkvm();

-

- // Process is done running for now.

- // It should have changed its p->state before coming back.

+

+ //PAGEBREAK: 42

+ // Per-CPU process scheduler.

+ // Each CPU calls scheduler() after setting itself up.

+ // Scheduler never returns. It loops, doing:

+ // - choose a process to run

+ // - swtch to start running that process

+ // - eventually that process transfers control

+ // via swtch back to the scheduler.

+ void

+ scheduler(void)

+ {

+ struct proc \*p;

+ struct cpu \*c = mycpu();

c->proc = 0;

+

+ for(;;){

+ // Enable interrupts on this processor.

+ sti();

+

+ // Loop over process table looking for process to run.

+ acquire(&ptable.lock);

+ for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

+ if(p->state != RUNNABLE)

+ continue;

+

+ // Switch to chosen process. It is the process's job

+ // to release ptable.lock and then reacquire it

+ // before jumping back to us.

+ c->proc = p;

+ switchuvm(p);

+ p->state = RUNNING;

+

+ swtch(&(c->scheduler), p->context);

+ switchkvm();

+

+ // Process is done running for now.

+ // It should have changed its p->state before coming back.

+ c->proc = 0;

+ }

+ release(&ptable.lock);

+

+ }

}

- release(&ptable.lock);

-

- }

-}

-

-// Enter scheduler. Must hold only ptable.lock

-// and have changed proc->state. Saves and restores

-// intena because intena is a property of this

-// kernel thread, not this CPU. It should

-// be proc->intena and proc->ncli, but that would

-// break in the few places where a lock is held but

-// there's no process.

-void

-sched(void)

-{

- int intena;

- struct proc \*p = myproc();

-

- if(!holding(&ptable.lock))

- panic("sched ptable.lock");

- if(mycpu()->ncli != 1)

- panic("sched locks");

- if(p->state == RUNNING)

- panic("sched running");

- if(readeflags()&FL\_IF)

- panic("sched interruptible");

- intena = mycpu()->intena;

- swtch(&p->context, mycpu()->scheduler);

- mycpu()->intena = intena;

-}

-

-// Give up the CPU for one scheduling round.

-void

-yield(void)

-{

- acquire(&ptable.lock); //DOC: yieldlock

- myproc()->state = RUNNABLE;

- sched();

- release(&ptable.lock);

-}

-

-// A fork child's very first scheduling by scheduler()

-// will swtch here. "Return" to user space.

-void

-forkret(void)

-{

- static int first = 1;

- // Still holding ptable.lock from scheduler.

- release(&ptable.lock);

-

- if (first) {

- // Some initialization functions must be run in the context

- // of a regular process (e.g., they call sleep), and thus cannot

- // be run from main().

- first = 0;

- iinit(ROOTDEV);

- initlog(ROOTDEV);

- }

-

- // Return to "caller", actually trapret (see allocproc).

-}

-

-// Atomically release lock and sleep on chan.

-// Reacquires lock when awakened.

-void

-sleep(void \*chan, struct spinlock \*lk)

-{

- struct proc \*p = myproc();

-

- if(p == 0)

- panic("sleep");

-

- if(lk == 0)

- panic("sleep without lk");

-

- // Must acquire ptable.lock in order to

- // change p->state and then call sched.

- // Once we hold ptable.lock, we can be

- // guaranteed that we won't miss any wakeup

- // (wakeup runs with ptable.lock locked),

- // so it's okay to release lk.

- if(lk != &ptable.lock){ //DOC: sleeplock0

- acquire(&ptable.lock); //DOC: sleeplock1

- release(lk);

- }

- // Go to sleep.

- p->chan = chan;

- p->state = SLEEPING;

-

- sched();

-

- // Tidy up.

- p->chan = 0;

-

- // Reacquire original lock.

- if(lk != &ptable.lock){ //DOC: sleeplock2

- release(&ptable.lock);

- acquire(lk);

- }

-}

-

-//PAGEBREAK!

-// Wake up all processes sleeping on chan.

-// The ptable lock must be held.

-static void

-wakeup1(void \*chan)

-{

- struct proc \*p;

-

- for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)

- if(p->state == SLEEPING && p->chan == chan)

- p->state = RUNNABLE;

-}

-

-// Wake up all processes sleeping on chan.

-void

-wakeup(void \*chan)

-{

- acquire(&ptable.lock);

- wakeup1(chan);

- release(&ptable.lock);

-}

-

-// Kill the process with the given pid.

-// Process won't exit until it returns

-// to user space (see trap in trap.c).

-int

-kill(int pid)

-{

- struct proc \*p;

-

- acquire(&ptable.lock);

- for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

- if(p->pid == pid){

- p->killed = 1;

- // Wake process from sleep if necessary.

- if(p->state == SLEEPING)

- p->state = RUNNABLE;

+

+ // Enter scheduler. Must hold only ptable.lock

+ // and have changed proc->state. Saves and restores

+ // intena because intena is a property of this

+ // kernel thread, not this CPU. It should

+ // be proc->intena and proc->ncli, but that would

+ // break in the few places where a lock is held but

+ // there's no process.

+ void

+ sched(void)

+ {

+ int intena;

+ struct proc \*p = myproc();

+

+ if(!holding(&ptable.lock))

+ panic("sched ptable.lock");

+ if(mycpu()->ncli != 1)

+ panic("sched locks");

+ if(p->state == RUNNING)

+ panic("sched running");

+ if(readeflags()&FL\_IF)

+ panic("sched interruptible");

+ intena = mycpu()->intena;

+ swtch(&p->context, mycpu()->scheduler);

+ mycpu()->intena = intena;

+ }

+

+ // Give up the CPU for one scheduling round.

+ void

+ yield(void)

+ {

+ acquire(&ptable.lock); //DOC: yieldlock

+ myproc()->state = RUNNABLE;

+ sched();

release(&ptable.lock);

- return 0;

}

- }

- release(&ptable.lock);

- return -1;

-}

-

-//PAGEBREAK: 36

-// Print a process listing to console. For debugging.

-// Runs when user types ^P on console.

-// No lock to avoid wedging a stuck machine further.

-void

-procdump(void)

-{

- static char \*states[] = {

- [UNUSED] "unused",

- [EMBRYO] "embryo",

- [SLEEPING] "sleep ",

- [RUNNABLE] "runble",

- [RUNNING] "run ",

- [ZOMBIE] "zombie"

- };

- int i;

- struct proc \*p;

- char \*state;

- uint pc[10];

-

- for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

- if(p->state == UNUSED)

- continue;

- if(p->state >= 0 && p->state < NELEM(states) && states[p->state])

- state = states[p->state];

- else

- state = "???";

- cprintf("%d %s %s", p->pid, state, p->name);

- if(p->state == SLEEPING){

- getcallerpcs((uint\*)p->context->ebp+2, pc);

- for(i=0; i<10 && pc[i] != 0; i++)

- cprintf(" %p", pc[i]);

+

+ // A fork child's very first scheduling by scheduler()

+ // will swtch here. "Return" to user space.

+ void

+ forkret(void)

+ {

+ static int first = 1;

+ // Still holding ptable.lock from scheduler.

+ release(&ptable.lock);

+

+ if (first) {

+ // Some initialization functions must be run in the context

+ // of a regular process (e.g., they call sleep), and thus cannot

+ // be run from main().

+ first = 0;

+ iinit(ROOTDEV);

+ initlog(ROOTDEV);

+ }

+

+ // Return to "caller", actually trapret (see allocproc).

+ }

+

+ // Atomically release lock and sleep on chan.

+ // Reacquires lock when awakened.

+ void

+ sleep(void \*chan, struct spinlock \*lk)

+ {

+ struct proc \*p = myproc();

+

+ if(p == 0)

+ panic("sleep");

+

+ if(lk == 0)

+ panic("sleep without lk");

+

+ // Must acquire ptable.lock in order to

+ // change p->state and then call sched.

+ // Once we hold ptable.lock, we can be

+ // guaranteed that we won't miss any wakeup

+ // (wakeup runs with ptable.lock locked),

+ // so it's okay to release lk.

+ if(lk != &ptable.lock){ //DOC: sleeplock0

+ acquire(&ptable.lock); //DOC: sleeplock1

+ release(lk);

+ }

+ // Go to sleep.

+ p->chan = chan;

+ p->state = SLEEPING;

+

+ sched();

+

+ // Tidy up.

+ p->chan = 0;

+

+ // Reacquire original lock.

+ if(lk != &ptable.lock){ //DOC: sleeplock2

+ release(&ptable.lock);

+ acquire(lk);

+ }

+ }

+

+ //PAGEBREAK!

+ // Wake up all processes sleeping on chan.

+ // The ptable lock must be held.

+ static void

+ wakeup1(void \*chan)

+ {

+ struct proc \*p;

+

+ for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)

+ if(p->state == SLEEPING && p->chan == chan)

+ p->state = RUNNABLE;

+ }

+

+ // Wake up all processes sleeping on chan.

+ void

+ wakeup(void \*chan)

+ {

+ acquire(&ptable.lock);

+ wakeup1(chan);

+ release(&ptable.lock);

+ }

+

+ // Kill the process with the given pid.

+ // Process won't exit until it returns

+ // to user space (see trap in trap.c).

+ int

+ kill(int pid)

+ {

+ struct proc \*p;

+

+ acquire(&ptable.lock);

+ for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

+ if(p->pid == pid){

+ p->killed = 1;

+ // Wake process from sleep if necessary.

+ if(p->state == SLEEPING)

+ p->state = RUNNABLE;

+ release(&ptable.lock);

+ return 0;

+ }

+ }

+ release(&ptable.lock);

+ return -1;

+ }

+

+ //PAGEBREAK: 36

+ // Print a process listing to console. For debugging.

+ // Runs when user types ^P on console.

+ // No lock to avoid wedging a stuck machine further.

+ void

+ procdump(void)

+ {

+ static char \*states[] = {

+ [UNUSED] "unused",

+ [EMBRYO] "embryo",

+ [SLEEPING] "sleep ",

+ [RUNNABLE] "runble",

+ [RUNNING] "run ",

+ [ZOMBIE] "zombie"

+ };

+ int i;

+ struct proc \*p;

+ char \*state;

+ uint pc[10];

+

+ for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

+ if(p->state == UNUSED)

+ continue;

+ if(p->state >= 0 && p->state < NELEM(states) && states[p->state])

+ state = states[p->state];

+ else

+ state = "???";

+ cprintf("%d %s %s", p->pid, state, p->name);

+ if(p->state == SLEEPING){

+ getcallerpcs((uint\*)p->context->ebp+2, pc);

+ for(i=0; i<10 && pc[i] != 0; i++)

+ cprintf(" %p", pc[i]);

+ }

+ cprintf("\n");

+ }

}

- cprintf("\n");

- }

-}