Coordination and more processes

Required reading: remainder of proc.c, sys_exec, sys_sbrk, sys_wait, sys_exit, and sys_kill.

Overview

Big picture: more programs than processors. How to share the limited number of processors among the programs? Last lecture covered basic mechanism: threads and the distinction between process and thread. Today expand: how to coordinate the interactions between threads explicitly, and some operations on processes.

Sequence coordination. This is a diferrent type of coordination than mutual-exclusion coordination (which has its goal to make atomic actions so that threads don't interfere). The goal of sequence coordination is for threads to coordinate the sequences in which they run.

For example, a thread may want to wait until another thread terminates. One way to do so is to have the thread run periodically, let it check if the other thread terminated, and if not give up the processor again. This is wasteful, especially if there are many threads.

With primitives for sequence coordination one can do better. The thread could tell the thread manager that it is waiting for an event (e.g., another thread terminating). When the other thread terminates, it explicitly wakes up the waiting thread. This is more work for the programmer, but more efficient.

Sequence coordination often interacts with mutual-exclusion coordination, as we will see below.

The operating system literature has a rich set of primivites for sequence coordination. We study a very simple version of condition variables in xv6: sleep and wakeup, with a single lock.

xv6 code examples

Sleep and wakeup - usage

Let's consider implementing a producer/consumer queue (like a pipe) that can be used to hold a single non-null char pointer:

```
struct pcq {
    void *ptr;
};

void*
pcqread(struct pcq *q)
```

```
{
    void *p;

    while((p = q->ptr) == 0)
        ;
    q->ptr = 0;
    return p;
}

void
pcqwrite(struct pcq *q, void *p)
{
    while(q->ptr != 0)
        ;
    q->ptr = p;
}
```

Easy and correct, at least assuming there is at most one reader and at most one writer at a time.

Unfortunately, the while loops are inefficient. Instead of polling, it would be great if there were primitives saying ``wait for some event to happen" and ``this event happened". That's what sleep and wakeup do.

Second try:

```
void*
pcqread(struct pcq *q)
{
    void *p;

    if(q->ptr == 0)
        sleep(q);
    p = q->ptr;
    q->ptr = 0;
    wakeup(q);  /* wake pcqwrite */
    return p;
}

void
pcqwrite(struct pcq *q, void *p)
{
    if(q->ptr != 0)
        sleep(q);
    q->ptr = p;
    wakeup(q);  /* wake pcqread */
    return p;
}
```

That's better, but there is still a problem. What if the wakeup happens between the check in the if and the call to sleep?

Add locks:

```
struct pcq {
    void *ptr;
    struct spinlock lock;
};

void*
pcqread(struct pcq *q)
{
    void *p;

    acquire(&q->lock);
    if(q->ptr == 0)
        sleep(q, &q->lock);
    p = q->ptr;
    q->ptr = 0;
    wakeup(q);  /* wake pcqwrite */
    release(&q->lock);
    return p;
}

void
pcqwrite(struct pcq *q, void *p)
{
    acquire(&q->lock);
    if(q->ptr != 0)
        sleep(q, &q->lock);
    q->ptr = p;
    wakeup(q);  /* wake pcqread */
    release(&q->lock);
    return p;
}
```

This is okay, and now safer for multiple readers and writers, except that wakeup wakes up everyone who is asleep on chan, not just one guy. So some of the guys who wake up from sleep might not be cleared to read or write from the queue. Have to go back to looping:

```
struct pcq {
    void *ptr;
    struct spinlock lock;
};

void*
pcqread(struct pcq *q)
{
    void *p;

    acquire(&q->lock);
    while(q->ptr == 0)
        sleep(q, &q->lock);
    p = q->ptr;
    q->ptr = 0;
    wakeup(q);  /* wake pcqwrite */
    release(&q->lock);
    return p;
}

void
```

```
pcqwrite(struct pcq *q, void *p)
{
    acquire(&q->lock);
    while(q->ptr != 0)
        sleep(q, &q->lock);
    q->ptr = p;
    wakeup(q);  /* wake pcqread */
    release(&q->lock);
    return p;
}
```
The difference between this an our original is that the body of the while loop is a much more efficient way to pause.

Now we've figured out how to use it, but we still need to figure out how to implement it.

Sleep and wakeup - implementation

Simple implementation:

```
void
sleep(void *chan, struct spinlock *lk)
{
    struct proc *p = curproc[cpu()];

    release(lk);
    p->chan = chan;
    p->state = SLEEPING;
    sched();
}

void
wakeup(void *chan)
{
    for(each proc p) {
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;
    }
}
```

What's wrong? What if the wakeup runs right after the release(lk) in sleep? It still misses the sleep.

Move the lock down:

```
void
sleep(void *chan, struct spinlock *lk)
{
    struct proc *p = curproc[cpu()];

    p->chan = chan;
    p->state = SLEEPING;
    release(lk);
    sched();
```

```
}

void
wakeup(void *chan)
{
    for(each proc p) {
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;
    }
}
```

This almost works. Recall from last lecture that we also need to acquire the proc_table_lock before calling sched, to protect p->jmpbuf.

```
void
sleep(void *chan, struct spinlock *lk)
{
    struct proc *p = curproc[cpu()];

    p->chan = chan;
    p->state = SLEEPING;
    acquire(&proc_table_lock);
    release(lk);
    sched();
}
```

The problem is that now we're using lk to protect access to the p->chan and p->state variables but other routines besides sleep and wakeup (in particular, proc_kill) will need to use them and won't know which lock protects them. So instead of protecting them with lk, let's use proc_table_lock:

```
void
sleep(void *chan, struct spinlock *lk)
{
    struct proc *p = curproc[cpu()];

    acquire(&proc_table_lock);
    release(lk);
    p->chan = chan;
    p->state = SLEEPING;
    sched();
}
void
wakeup(void *chan)
{
    acquire(&proc_table_lock);
    for(each proc p) {
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;
    }
    release(&proc_table_lock);
}
```

One could probably make things work with lk as above, but the relationship between data and locks would be more complicated with no real benefit. Xv6 takes the easy way out and says that elements in the proc structure are always protected by proc_table_lock.

Use example: exit and wait

If proc_wait decides there are children to be waited for, it calls sleep at line 2462. When a process exits, we proc_exit scans the process table to find the parent and wakes it at 2408.

Which lock protects sleep and wakeup from missing each other? Proc_table_lock. Have to tweak sleep again to avoid double-acquire:

```
if(lk != &proc_table_lock) {
    acquire(&proc_table_lock);
    release(lk);
}
```

New feature: kill

Proc_kill marks a process as killed (line 2371). When the process finally exits the kernel to user space, or if a clock interrupt happens while it is in user space, it will be destroyed (line 2886, 2890, 2912).

Why wait until the process ends up in user space?

What if the process is stuck in sleep? It might take a long time to get back to user space. Don't want to have to wait for it, so make sleep wake up early (line 2373).

This means all callers of sleep should check whether they have been killed, but none do. Bug in xv6.

System call handlers

Sheet 32

Fork: discussed copyproc in earlier lectures. Sys_fork (line 3218) just calls copyproc and marks the new proc runnable. Does fork create a new process or a new thread? Is there any shared context?

Exec: we'll talk about exec later, when we talk about file systems.

Sbrk: Saw growproc earlier. Why setupsegs before returning?