

xv6©-rev10
(Copyright Frans Kaashoek, Robert Morris, and Russ Cox.)
Events

Carmi Merimovich

Tel-Aviv Academic College

January 10, 2017

Recall:

- Switching to the **scheduler**.
- Switching to a **process**.

sched()

```
2808 void sched(void) {
    int intena;

    if (!holding(&ptable.lock))
        panic("sched_ptable.lock");
    if (mycpu()->ncli != 1)
        panic("sched_locks");
    if (myproc()->state == RUNNING)
        panic("sched_running");
    if (readeflags() & FL_IF)
        panic("sched_interruptible");

    intena = mycpu()->intena;
    swtch(&myproc()->context, mycpu()->scheduler);
    mycpu()->intena = intena;
}
```

scheduler()

2758

```
void scheduler(void) {  
    struct proc *p;  
    for (;;) { sti();  
        mycpu()->proc=0  
        acquire(&ptable.lock);  
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {  
            if (p->state != RUNNABLE) continue;  
  
            mycpu()->proc = p;  
            switchvm(p);  
            p->state = RUNNING;  
            switch(&(mycpu()->scheduler), p->context);  
            switchkvm();  
  
            mycpu()->proc = 0;  
        }  
        release(&ptable.lock);  
    }  
}
```

The **unnamed** event.

(does not exists in xv6)

Process suspension code

Consider the following code snippet:

```
suspend () {  
    myproc()->state = SLEEPING;  
    sched ();  
}
```

- What will happen upon execution?
 - The process will get stuck.
- How do we unstuck it?
 - A **Different** process should un-stuck it.
 - Meaning setting its **state** to **RUNNABLE**.

un-suspending

```
void unsuspend() {  
    struct proc *p;  
    for (p=ptable.proc; p < &ptable.proc[NPROC] ;p++) {  
        if (p->state == SLEEPING)  
            p->state = RUNNABLE;  
    }  
}
```

- What happens upon execution of **unsuspend()**:
 - **All** suspended processes become **RUNNABLE**.
 - Note. The processes will actually run at the discretion of the scheduler.

Example: Using the unnamed event.

Waiting for a Zombie'd child

```
struct proc *child_kaput() {  
    for(;;) { int nochild=1;  
        acquire(&ptable.lock);  
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {  
            if (p->state == UNUSED || p->parent != myproc()) continue;  
            if (p->state == ZOMBIE) {  
                release(&ptable.lock);  
                return (p);  
            }  
            nochild=0;  
        }  
        if (nochild) {  
            release(&ptable.lock);  
            return (0);  
        }  
        suspend();    // What is expected of the child????  
        release(&ptable.lock)  
    } // Is this any good????
```

Waiting for a Zombie'd child

```
struct proc *child_kaput() {
    for(;;) { int nochild=1;
        acquire(&ptable.lock);
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if (p->state == UNUSED || p->parent != myproc()) continue;
            if (p->state == ZOMBIE) {
                release(&ptable.lock);
                return (p);
            }
        }
        nochild=0;
    }
    if (nochild) {
        release(&ptable.lock);
        return (0);
    }
    if (myproc()->killed) {release(&ptable.lock; return(-1)
suspend();    // What is expected of the child????
release(&ptable.lock)
```

Zombie'ing a process

```
acquire(&ptable.lock)
:
:
myproc()->state=ZOMBIE;
unsuspend();
sched()
```

Comments on the unnamed event

- Each exiting process should call **unsuspend()**.
- Each parent waiting for terminating child should call **suspend()**.
- So a parent is unsuspended for each process exiting!
- In fact, if all we have is the unnamed event, any event unsuspends a parent.
- It would be nice if we reduce the unneeded unsuspends.

Named events.

Process suspension code

Consider the following code snippet:

```
suspend(int id) {  
    p->state = SLEEPING;  // No such state  
    p->chan = id;  
    sched();  
}
```

- What will happen upon execution?
 - The process will stuck.
- How do we unstuck it?
 - A **Different** process should un-stuck it with the **right id**.

un-suspending

```
void unsuspend(int id) {  
    struct proc *p;  
    for (p=ptable.proc; p < &ptable.proc[NPROC] ;p++) {  
        if (p->state == SLEEPING && p->chan == id)  
            p->state = RUNNABLE;  
    }  
}
```

- What happens upon execution of **unsuspend(id)**:
 - **All** suspended processes with id are runnable.
 - Note. The processes will actually run at the discretion of the scheduler.

How do we choose an id?

- We can make a whole system of id's.
- Or have a simple .h file.
- Or use a person capable of being the id's tzar.
 - E.g., John Postel (alas, RFC 2468).
- Or we can we Thompson's trick.

Waiting for a Zombie'd child

```
struct proc *child_kaput() {
    for(;;) { int nochild=1;
        acquire(&ptable.lock);
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if (p->state == UNUSED || p->parent != myproc()) continue;
            if (p->state == ZOMBIE) {
                release(&ptable.lock);
                return (p);
            }
            nochild = 0;
        }
        if (nochild) {
            release(&ptable.lock);
            return (0);
        }
        if (myproc()->killed) { release(&ptable.lock); return(-1);
        suspend(myproc()); //unsuspend(myproc()->parent).
        release(&ptable.lock);
    }
}
```

Zombie'ing a process

```
acquire(&ptable.lock)
:
:
myproc()->state=ZOMBIE;
unsuspend(myproc()->parent);
sched()
```

Comments on named event

- The number of unneeded unsuspends reduced considerably.
- We cannot be **sure** no other event uses the same id, but it is plausible.
- So, we should still assume unsuspending for no good reason can happen.

What about locks?

- **child_kaput** has a lock on **ptable.lock**.
- **exit** (i.e., zombie'ing) has a lock on **ptable.lock**.
- This situation is quite common.
- However, different spinlocks are also plausible.

suspend with spinlock

```
suspend(int id, struct spinlock *lk) {  
    if (lk != ptable.lock) {  
        acquire(&ptable.lock);  
        release(lk);  
    }  
    myproc()->state = SLEEPING;    // No such state  
    myproc()->chan = id;  
    sched();  
    if (lk != ptable.lock) {  
        release(&ptable.lock);  
        acquire(lk);  
    }  
}
```

Waiting for a Zombie'd child

```
struct proc *child_kaput() {
    for(;;) { int nochild=1;
        acquire(&ptable.lock);
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if (p->state == UNUSED || p->parent != myproc()) continue;
            if (p->state == ZOMBIE) {
                release(&ptable.lock);
                return (p);
            }
            nochild = 0;
        }
        if (nochild) {
            release(&ptable.lock);
            return (0);
        }
        if (myproc()->killed) { release(&ptable.lock); return(-1);
        suspend(myproc(), &ptable.lock); //unsuspend(proc->parent)
        release(&ptable.lock);
    }
}
```

xv6 event management

sleep

```
2874 void sleep(void *chan, struct spinlock *lk) {  
    if (lk != &ptable.lock) {  
        acquire(&ptable.lock);  
        release(lk);  
    }  
    myproc()->chan = chan;  
    myproc()->state = SLEEPING;  
    sched();  
    myproc()->chan = 0;  
  
    if (lk != &ptable.lock) {  
        release(&ptable.lock);  
        acquire(lk);  
    }  
}
```


Using **sleep**

- **sleep** is called always (always!) inside a loop.
- Resuming after **sleep** means an event has occurred.
- Not necessarily the intended event!
- **myproc()->killed** is almost always(!!) checked before entering **sleep**.
- Only in the file-system **sleep** is called without **myproc()->killed** checking.

wakeup

- **wakeup** is the “event manager” of xv6.
- **wakeup(chan)** scans the **proc** table and switches to the **RUNNABLE** state all processes which are **SLEEPING** and their **chan** field is equal to the **chan** argument.
- **wakeup** protects the scanning with **ptable.lock**.
- This is in accordance with the **proc** table scanning contract.
- **wakeup1** has the same functionality as **wakeup** without the **ptable.lock** protection.
- **wakeup1** should be used (must be used!) if **ptable.lock** was already acquired!

wakeup

```
2953 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;  
}  
  
2964 void wakeup(void *chan) {  
    acquire(&ptable.lock);  
    wakeup1(chan);  
    release(&ptable.lock);  
}
```