Sleep and wakeup in xv6

Mythili Vutukuru

CSE, IIT Bombay

Sleep and wakeup in xv6 (1)

- xv6 does not have userspace threads, only single threaded processes
- But multiple processes may be in kernel mode on different CPU
  - Uses locks to protect access to shared kernel data structures
- OS also needs a mechanism to let processes sleep (e.g., when process makes blocking disk read syscall) and wakeup when some events occur (e.g., disk has raised interrupt and data is ready)
- Process P1 in kernel mode calls sleep to give up CPU, gets blocked until event
- Another process P2 (in kernel mode) wakes up P1 when the event occurs

Sleep and wakeup in xv6 (2)

- A process P1 that wishes to block and give up CPU calls “sleep”
  - Example: process reads a block from disk, must block until disk read completes
  - Read syscall → sleep → sched() to give up CPU
- Another process P2 calls “wakeup” when event to unblock P1 occurs
  - P2 calls wakeup → marks P1 as runnable, no context switch immediately
  - Example: disk interrupt occurred when P2 is running, P2 runs interrupt handler, which will call wakeup
- P1 will be scheduled at a later time, will resume at sched(), return
- Spinlock protects atomicity of sleep: P1 calls sleep with some spinlock L held, P2 calls wakeup with same spinlock L held

Sleep and wakeup in xv6 (3)

- How does P2 know which process to wake up?
- When P1 sleeps, it sets a channel (void * chan) in its struct proc
  - Arguments to sleep: channel, spinlock to protect atomicity of sleep
- P2 calls wakeup on same channel
  - Arguments to wakeup: channel (lock must be held)
- Channel = any value known to both P1 and P2
  - Example: channel value for disk read can be address of disk block

Example: wait and exit

- If wait called in parent while children are running, parent calls sleep and gives up CPU (channel is parent struct proc pts, lock is ptable.lock)

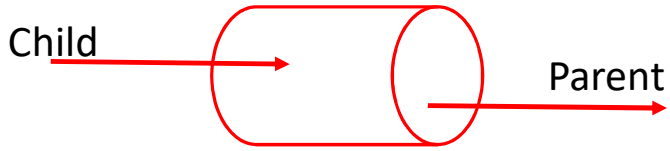
```
2706    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2707    sleep(curproc, &ptable.lock);
```

- In exit, child acquires ptable.lock, wakes up parent using its channel

```
2650  // Parent might be sleeping in wait().
2651  wakeup1(curproc->parent);
```

- Why is terminated process memory cleaned up by parent?
  - When a process calls exit, kernel stack, page table etc are in use, all this memory cannot be cleared until terminated process has been taken off the CPU

Example: pipes in xv6 (1)



- xv6 provides anonymous pipes for IPC between parent and child processes
- Example: Parent P and child C share anonymous pipe
- Child C writes into pipe, parent P reads from pipe
- One of P or C closes read end, other closes write end

```
//userspace code

int fd[2]
pipe(fd) //syscall to create pipe

int ret = fork()

if(ret == 0) {//child
   close(fd[0]) //close read end
   write(fd[1], message, ..)
}
else {//parent
   close(fd[1]) //close write end
   read(fd[0], message, ..)
}
```

Example: pipes in xv6 (2)

- Internal implementation inside kernel
  - Common shared buffer, protected by a spinlock
  - Write system call stores data in shared buffer
  - Read system call returns data from shared buffer
  - Variables nread and nwrite indicate number of bytes read/written in buffer

```
6762 struct pipe {
6763   struct spinlock lock;
6764   char data[PIPESIZE];
6765   uint nread;     // number of bytes read
6766   uint nwrite;    // number of bytes written
6767   int readopen;   // read fd is still open
6768   int writeopen;  // write fd is still open
6769 };
```

Example: pipes in xv6 (3)

- Implementation of pipe read and write system calls uses sleep/wakeup
- Pipe reader sleeps if pipe is empty, pipe writer wakes it up
- Pipe writer sleeps if pipe is full, pipe reader wakes it up
- Channel for sleep/wakeup = address of pipe structure variables

```
6829 int
6830 pipewrite(struct pipe *p, char *addr, int n)
6831 {
6832   int i;
6833
6834   acquire(&p->lock);
6835   for(i = 0; i < n; i++){
6836     while(p->nwrite == p->nread + PIPESIZE){
6837       if(p->readopen == 0 || myproc()->killed){
6838         release(&p->lock);
6839         return -1;
6840       }
6841       wakeup(&p->nread);
6842       sleep(&p->nwrite, &p->lock);
6843     }
6844     p->data[p->nwrite++ % PIPESIZE] = addr[i];
6845   }
6846   wakeup(&p->nread);
6847   release(&p->lock);
6848   return n;
6849 }
```

pipe is full

writer's channel for sleep is address of nwrite variable

Example: pipes in xv6 (4)

```
6850 int
6851 piperead(struct pipe *p, char *addr, int n)
6852 {
6853   int i;
6854
6855   acquire(&p->lock);
6856   while(p->nread == p->nwrite && p->writeopen){
6857     if(myproc()->killed){
6858       release(&p->lock);
6859       return -1;
6860     }
6861     sleep(&p->nread, &p->lock);
6862   }
6863   for(i = 0; i < n; i++){
6864     if(p->nread == p->nwrite)
6865       break;
6866     addr[i] = p->data[p->nread++ % PIPESIZE];
6867   }
6868   wakeup(&p->nwrite);
6869   release(&p->lock);
6870   return i;
6871 }
```

pipe is empty

reader's channel is address of nread variable

pipe lock protects atomicity of sleep

Sleep function

```
2873 void
2874 sleep(void *chan, struct spinlock *lk)
2875 {
2876   struct proc *p = myproc();
2877
2878   if(p == 0)
2879     panic("sleep");
2880
2881   if(lk == 0)
2882     panic("sleep without lk");
2883
2884   // Must acquire ptable.lock in order to
2885   // change p->state and then call sched.
2886   // Once we hold ptable.lock, we can be
2887   // guaranteed that we won't miss any wakeup
2888   // (wakeup runs with ptable.lock locked),
2889   // so it's okay to release lk.
2890   if(lk != &ptable.lock){
2891     acquire(&ptable.lock);
2892     release(lk);
2893   }
2894   // Go to sleep.
2895   p->chan = chan;
2896   p->state = SLEEPING;
2897
2898   sched();
2899
2900   // Tidy up.
2901   p->chan = 0;
2902
2903   // Reacquire original lock.
2904   if(lk != &ptable.lock){
2905     release(&ptable.lock);
2906     acquire(lk);
2907   }
2908 }
```

- Sleep and wakeup called by processes with same lock held (to protect atomicity of sleep)
- Acquire ptable lock (if not already taken), then release other spinlock
- Reacquire original lock on return

Wakeup function

- Wakeup acquires ptable.lock to change process to runnable
- If lock protecting atomicity of sleep is ptable.lock itself, then directly call wakeup1
- Wakes up all processes sleeping on a channel in ptable (more like signal broadcast of condition variables)

```
2950 // Wake up all processes sleeping on chan.
2951 // The ptable lock must be held.
2952 static void
2953 wakeup1(void *chan)
2954 {
2955   struct proc *p;
2956
2957   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2958     if(p->state == SLEEPING && p->chan == chan)
2959       p->state = RUNNABLE;
2960 }
2961
2962 // Wake up all processes sleeping on chan.
2963 void
2964 wakeup(void *chan)
2965 {
2966   acquire(&ptable.lock);
2967   wakeup1(chan);
2968   release(&ptable.lock);
2969 }
```