日曜研究室

技術的な観点から日常を綴ります

[xv6 #48] Chapter 4 – Scheduling – Code: Sleep and wakeup

テキストの58〜59ページ

本文

xv6におけるsleepとwakeupの実装について見てみよう。
sleepに現在のプロセスをSLEEPINGとして印をつけさせ、そしてschedを呼びそのプロセッサを解放し、wakeupは与えられたポインタ上でスリープ中のプロセスを探し、それをRUNNABLEとして印をつける、というのが基本的なアイデアである。

sleepはいくつかの条件チェックから始まる。
現在のプロセスがあり（proc == 0 だとpanic）、ロックを渡されてなければ（lk == 0 でもpanic）ならない。
チェックが終わるとまず、ptable.lockを獲得する。
そのままでは、ptable.lockとlkの両方を保持したままスリープしてしまう。
lkの保持は、（例えばrecvのような）呼び出し側で必要とされる。
lkを保持しているということは、wakeup(chan)を呼びはじめれるプロセス（例えばsendを実行してるプロセス）が他にないということを確実にする。
この時点でsleepはptable.lockを保持し、lkを安全に解放出来るようになる。
他のいくつかのプロセスが、wakeup(chan)を呼び出すかもしれないが、wakeupはptable.lockが獲得可能になるまで実行されないので、sleepがプロセスを確実にスリープさせるまで待たなければならず、起き損ないの問題（前節参照）は起きない。

proc.cのsleep関数

```
01 // Atomically release lock and sleep on chan.
02 // Reacquires lock when awakened.
03 void
04 sleep(void *chan, struct spinlock *lk)
05 {
06   if(proc == 0)
07     panic("sleep");
```

```
08
09    if(lk == 0)
10      panic("sleep without lk");
11
12    // Must acquire ptable.lock in order to
13    // change p->state and then call sched.
14    // Once we hold ptable.lock, we can be
15    // guaranteed that we won't miss any wakeup
16    // (wakeup runs with ptable.lock locked),
17    // so it's okay to release lk.
18    if(lk != &ptable.lock){  //DOC: sleeplock0
19      acquire(&ptable.lock);  //DOC: sleeplock1
20      release(lk);
21    }
22
23    // Go to sleep.
24    proc->chan = chan;
25    proc->state = SLEEPING;
26    sched();
27
28    // Tidy up.
29    proc->chan = 0;
30
31    // Reacquire original lock.
32    if(lk != &ptable.lock){  //DOC: sleeplock2
33      release(&ptable.lock);
34      acquire(lk);
35    }
36  }
```

微妙に複雑な場合がある。
もしlkが&ptable.lockと等しい場合、sleepはそれを&ptable.lockとして獲得し、lkとして解放しようとしてデッドロックに陥るだろう。
この場合、sleepはacquireとreleseがお互いに相殺されることを考慮し、それらを完全にスキップする。

この時点で、sleepはptable.lockのみを保持し、スリープチャンネル記録し、プロセスの状態を変更し、schedを呼ぶことで、プロセスをスリープさせる事ができる。

後のどこかの段階で、プロセスはwakeup(chan)を呼ぶだろう。
wakeupはptable.lockを獲得し、実際の処理を行うwakeup1を呼ぶ。
wakeup1はプロセスの状態を操作するし、前節で見たように、ptable.lockはsleepとwakeupがお互いに機会を逃さないようにすることを確実にするので、wakeupがptable.lockを保持する事はとても重要である。
ときおり、スケジューラはptable.lockを既に保持しているときにwakeupを実行する必要があるので、wakeup1として分離されている。
直接wakeup1を呼ぶ例は後で説明する。
wakeup1は、プロセステーブルを走査する。
chanが合致し状態がSLEEPINGなプロセスを見つけたら、そのプロセスの状態をRUNNABLEに変更する。
次にスケジューラが実行されたとき、そのプロセスは実行準備が整ってるとしてスケジューラに扱われるだろう。

proc.cのwakeup1, wakeup関数

```
01 // Wake up all processes sleeping on chan.
02 // The ptable lock must be held.
03 static void
04 wakeup1(void *chan)
05 {
06   struct proc *p;
07
08   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
09     if(p->state == SLEEPING && p->chan == chan)
10       p->state = RUNNABLE;
11 }
12
13 // Wake up all processes sleeping on chan.
14 void
15 wakeup(void *chan)
16 {
17   acquire(&ptable.lock);
18   wakeup1(chan);
19   release(&ptable.lock);
20 }
```

wakeupは常に、wakeupの条件がなんであれその監視を妨げるようなロックを保持してる間、呼ばれなければならない。（謎）
前節の例では、そのようなロックにあたるのはq->lockである。
スリープ中のプロセスがなぜ起き損ねたくないか、そのための完全な根拠は、スリープするまで条件をチェックする前から常に、条件上もしくはptable.lockもしくは両方のロックを保持する。（謎）
それらのロックの両方を保持してる間、wakeupが実行してから、wakeupはスリープさせる可能性があるものが条件をチェックする前、もしくはスリープさせる可能性があるものそれ自身がスリープした後に実行すべきである。（謎）

ときおり、複数のプロセスが同じチャンネル上でスリープしてる場合がある。
例えば、2つ以上のプロセスが1つのパイプから読み込む場合である。
1回のwakeupの呼び出しは、それらすべてを起こすだろう。
それらのひとつは、最初に実行されsleepを呼びロックを獲得し、そして（パイプの例だと）パイプに書きこまれた何らかのデータを読み込むだろう。
その他のプロセスは、起こされたにも関わらず、読み込むべきデータがないということに気づくだろう。
そういったプロセスの視点から見ると、そのwakeupは”見せかけ”であり、そういったプロセスはまたスリープする必要がある。
こんな理由で、sleepは常に、条件をチェックするループの中で呼ばれるようになっている。

sleepとwakeupの呼び出し側は、チャンネルとして何らかの共通の便利な番号を使うことが出来る。
習慣上、xv6は、ディスクバッファのような、待機に関連したカーネルのデータ構造のアドレスをしばしば使う。
もしsleep/wakeupを2箇所で使ってて、偶然にも同じチャンネルを選択してしまっても、害はない。
見せかけのwakeupに遭遇するだろうが、上記で説明したようなループは、この問題に耐性がある。

sleep/wakeupの魅力の多くは、両方共軽く（スリープチャンネルとして働くような特別なデータ構造を作る必要がない）、間接的なレイヤー（呼び出し側はsleep/wakeupが相互に影響しあう詳細な方法を知る必要がない）を提供するところにある。

感想

sleepとwakeupのxv6における実装の説明です。

今回ちょっと原文の言い回しが難しかったです。
途中かなり謎な段落もありました。

ぶっちゃけ今回は文章読まなくても、ソースと前節の内容を理解できてれば問題ないような気がします。

カテゴリー: 技術 l タグ: xv6 l 投稿日: 2012/3/22 木曜日 [http://peta.okechan.net/blog/archives/1561] l