日曜研究室

技術的な観点から日常を綴ります

[xv6 #47] Chapter 4 – Scheduling – Sleep and wakeup

テキストの55〜57ページ

本文

ロックは、CPU同士やプロセス同士がお互いに干渉し合うのを避けるのに役立ち、スケジューリングは、複数のプロセスがCPUを共有するのに役立つ。
しかしこれまで、我々はプロセス間通信を楽にするような概念は持ち合わせていなかった。
“スリープ”（sleep）と”ウェイクアップ”（wakeup）はその間隙を埋め、あるプロセスがあるイベントを待つためにスリープさせ、一度イベントが起きたら他のプロセスにそのプロセスをウェイクアップさせる事を可能にする。
スリープとウェイクアップは、しばしば順序協調（シーケンスコーディネーション, sequence coordination, オレオレ訳なので注意）、条件付き同期化（コンディショナルシンクロナイゼーション, conditional synchronization, こちらもオレオレ訳なので注意）の機構と呼ばれる。
OS学的に他の似たような機構はたくさんある。

我々が何を言いたいかハッキリさせるために、シンプルな生産者/消費者（producer/consumer）キューについて考えてみよう。
このキューは、プロセッサとデバイスドライバを同期化するためにIDEドライバによって使われたキューに似ている。（第2章参照）
しかし、抽象的なすべてのIDE特有のコードはない。
このキューは、ひとつのプロセスが他のプロセスにゼロじゃないポインタを送ることが出来る。
送信側と受信側はそれぞれひとつしかなく、それぞれ別のCPUで実行されていると仮定すると、以下の実装は正しい。

```
100 struct q {
101   void *ptr;
102 };
103
104 void*
105 send(struct q *q, void *p)
106 {
```

```
107   while(q->ptr != 0)
108     ;
109   q->ptr = p;
110 }
111
112 void*
113 recv(struct q *q)
114 {
115   void *p;
116
117   while((p = q->ptr) == 0)
118     ;
119   q->ptr = 0;
120   return p;
121 }
```

sendは、キューが空（ptr == 0）になるまでループし、そしてキューの中にポインタpをセットする。
recvは、キューに何か入ってる状態になるまでループし、ポインタを取り出す。
別のプロセスで実行されるとき、sendとrecvは両方ともq->ptrを編集するが、sendはq->ptrがゼロのときだけ書き込み、recvはq->ptrがゼロじゃないときだけ書き込むので、お互いに踏みつけることはない。

上記の実装は多分正しいが、効率が悪い。
送信側がたまにしか送信しない場合、受信側はポインタが来るのを期待してループしてる間、その時間のほとんどを消費してしまうだろう。
sendがポインタをセットしたときに受信側に通知する方法があれば、受信側のCPUは（その通知を待ってる間、他の）もっと生産的な仕事を見つけることができただろう。

次のように動作する、sleepとwakeupという関数の組について想像してみよう。
sleep(chan)は、ウェイトチャンネル（wait channel）と呼ばる任意の値であるchan上でスリープする。
sleepは、呼び出したプロセスをスリープさせ、他の仕事のためにCPUを解放する。
wakeup(chan)は、chan上でスリープしてるすべてのプロセス（もしあるなら）を起こし、それらのsleepの呼び出しを戻らせる。
chan上で待っているプロセスがない場合、wakeupは何もしない。
sleepとwakeupを使うことによって、我々はキューの実装を次のように改良出来る。

```
201 void*
202 send(struct q *q, void *p)
203 {
204   while(q->ptr != 0)
205     ;
206   q->ptr = p;
207   wakeup(q);  /* wake recv */
208 }
209
210 void*
211 recv(struct q * q)
212 {
213   void* p;
214
215   while((p = q->ptr) == 0)
```

```
216     sleep(q);
217   q->ptr = 0;
218   return p;
219 }
```

recvは、スピンする代わりにCPUを解放するようになった。素敵だ。
しかしながら、このインターフェイスによるsleepとwakeupの設計が直接的ではなく、起き損ない問題（lost wakeup probrem）として知られているものから被害を受けることが分かる。（図4-2参照）
recvが215行目でq->ptr == 0であるということを見つけ、sleepを呼ぶことを決定すると仮定しよう。
recvがsleep出来る前、sendは他のCPUで実行されていて、それはq->ptrを非ゼロに変更しwakeupを呼ぶ。
そのときwakeupはスリープ中のプロセスが無いので結果的に何もしない。
そしてrecvは216行目の実行を続行し、sleepを呼びスリープする。
これは問題を引き起こす。
recvはすでにポインタが届いてるのにスリープして待つ事になる。
次のsendは、recvがキューのポインタを取り出すのを待ち続け停止するだろう。
この時点でシステムはデッドロックに陥るだろう。

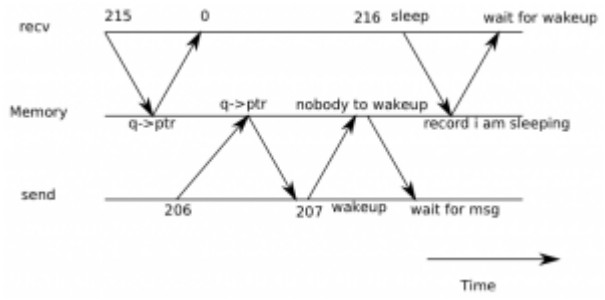

図4-2 起き損ない問題の例

この問題の根源は、recvはq->ptr == 0のときだけスリープしなければならないというインバリアントが、都合の悪い瞬間に実行されているsendによって侵害されるというところにある。
このインバリアントを保護するために、我々は、sleepを呼び出したプロセスがスリープしたあとにsleepによって解放されるようなロックを導入する。
これは、上記の例のような起き損ないを避ける。
一度sleepを呼び出したプロセスが再度起きたら、sleepは戻る前にそのロックを再獲得する。
次のコードがあり得るとしたい。

```
300 struct q {
301   struct spinlock lock;
302   void *ptr;
303 }
304
305 void*
306 send(struct q *q, void *p)
307 {
308   acquire(&q->lock);
309   while(q->ptr != 0)
310     ;
```

```
311   q->ptr = p;
312   wakeup(q);
313   release(&q->lock);
314 }
315
316 void*
317 recv(struct q *q)
318 {
319   void* p;
320
321   acquire(&q->lock);
322   while((p = q->ptr) == 0)
323     sleep(q, &q->lock);
324   q->ptr = 0;
325   release(&q->lock);
326   return p;
327 }
```

recvがq->lockを保持するという事実は、recvがq->ptrをチェックしsleepを呼ぶ間に、sendがwakeupを実行することを防ぐ。
もちろん、このままだと送信側のプロセスがwakeupを呼ぶのを妨害し、デッドロックに陥るので、受信側のプロセスは、スリープしてる間q->lockは保持しないほうがいい。
だから、sleepがアトミックにq->lockを開放し、プロセスを眠りに導くということが必要である。

完全な送信側/受信側の実装では、前回のsendで送られた値を受信側が取り出すのを待つときにも、send内でスリープするようになってるだろう。

感想

プロセス間通信と、その際の強力なツールになるsleepとwakeupについての話です。
重要なのは、ここではまだ実際のコードは一切出てきてないという事です。
実際のコードは多分次の節で出てきます。

最後の例では、変なタイミングでsleepやwakeupが実行されないようにロックが導入されてますが、そのままだとrecvがロックを保持してデータが届くのを待っている間、sendは何も出来ない（acquireで止まる）ということになります。
これを避けるために、sleep内では（本文ではコードは示されてませんが）通常の獲得→解放という順序とは逆に、q->lockを解放してから獲得するようになってるので、受信側がスリープ中（データ待ち中）はsendが実行可能になるということになります。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/21 水曜日 [http://peta.okechan.net/blog/archives/1554] |