Subject: Report Operating System Lecture Practice Synchronization
From: IKENOYA Katsutoshi <j05002@ie.u-ryukyu.ac.jp>
Date: Sun, 21 Jan 2007 13:56:26 +0900
To: Shinji KONO <kono@ie.u-ryukyu.ac.jp>

学籍番号：055702B

・開発環境
OS : Mac OS X 10.4.8
コンパイラ :gcc version 4.0.1 (Apple Computer, Inc. build 5250)
java version "1.5.0_06"

・実行環境
同上

課題1 相互排除を行わないプログラムの実行の考察

・実行結果

```
[j05002@6-synch]% ./mp-sem-mutex
process_A() done. shared_resource == 5.
process_B() done. shared_resource == 5.
[j05002@6-synch]%
```

プロセスA、プロセスBが交互に共有資源xを読み出し、xに１を加え、共有資源に
書き戻すという事を行っているので、本来ならば共有資源の値は、１０になって
いなければならないが結果では５となっている。
これは、プロセスAが共有資源に書き戻す前にプロセスBが共有資源を読み出して
いることから起こったと考えられる。

課題2 セマフォによる相互排除

・ソースの抜粋

ソースの場所：/home/y05/j05002/6-synch/mp-sem-mutex.c

省略

```
void *
process_A(void *arg)
{
int i ;
register int x ;

for( i=0 ; i<5 ; i++ )
{
pthread_mutex_lock(&mutex); /*変更点*/
x = shared_resource ; /* 共有資源の読み出し */
sched_yield();
x = x + 1 ; /* １を加える */
sched_yield();
shared_resource = x ; /* 共有資源に書き戻す */
sched_yield();
pthread_mutex_unlock(&mutex);/*変更点*/
}
printf("process_A() done. shared_resource == %d.¥n",
shared_resource );
}
```

```
void *
process_B(void *arg)
{
int i ;
register int x ;

for( i=0 ; i<5 ; i++ )
{
pthread_mutex_lock(&mutex);/*変更点*/
x = shared_resource ;
sched_yield();
x = x + 1 ;
sched_yield();
shared_resource = x ;
sched_yield();
pthread_mutex_unlock(&mutex);/*変更点*/
}
printf("process_B() done. shared_resource == %d.¥n",
shared_resource );
}
```

・実行結果

```
[j05002@6-synch]% ./mp-sem-mutex
process_A() done. shared_resource == 9.
process_B() done. shared_resource == 10.
```

実行結果より共有資源の値が１０となっており、
期待した通りの動作を行っていることが分かる。

課題3 生産者消費者問題

・ソースの抜粋

ソースの場所：/home/y05/j05002/6-synch/mp-sem-prodcons.c

省略

```
void *
consumer(void *arg) /* 消費者プロセス */
{

int x;
int i;
x = 0;
fprintf(stderr,"consumer(): started.¥n");
for( i=0 ; i<nloop; i++){
sem_p( &count);
sem_p(&mutex_c);

x = buffer[j_c] ;
fprintf(stderr,"consumer(): get %d.¥n",x );

buffer[i_p] = x;
j_c++ ;
if( j_c >= BUFFSIZE) j_c = 0;
sem_v( &mutex_c);
sem_v( &rem);
}

}
```

・実行結果

```
[j05002@6-synch]% ./mp-sem-prodcons
consumer(): started.
producer(): started.
producer(): put 0.
producer(): put 2.
producer(): put 4.
consumer(): get 0.
consumer(): get 2.
consumer(): get 4.
producer(): put 6.
producer(): put 8.
producer(): put 10.
consumer(): get 6.
consumer(): get 8.
consumer(): get 10.
producer(): put 12.
producer(): put 14.
producer(): put 16.
consumer(): get 12.
consumer(): get 14.
consumer(): get 16.
producer(): put 18.
consumer(): get 18.
[j05002@6-synch]%
```

生産者は空きができたことを確認し、情報を生産して、情報を入れたことを
消費者に伝える。
消費者は情報が入ったことを確認して、情報を消費し、情報に空きができた
ことを生産者に伝えている。

課題4 複数の生産者と複数の消費者

生産者と消費者の数がそれぞれ 2 の場合と、
生産者の数が 2 で、消費者の数が 3 の場合の二通りで実行してみた。

(1)生産者と消費者の数がそれぞれ 2

・ソースの抜粋

ソースの場所：/home/y05/j05002/6-synch/mp-sem-prodcons2-2.c

```
#include <stdio.h>
#include <pthread.h>
#ifdef sun
#define sched_yield thr_yield
#endif


void *consumer_A(void *);
void *producer_A(void *);
void *consumer_B(void *);
void *producer_B(void *);

pthread_t thread_consumer_A,thread_producer_A,
thread_consumer_B,thread_producer_B;
```

省略

```
int
main(int ac,char *av[])
{
/* mp_user_main() は、軽量プロセスを 4 個作り終了する。*/
```

```
j_c = 0 ;
i_p = 0 ;

sem_init( &mutex_p,1 ); /* 相互排除用のセマフォの初期化 */
sem_init( &mutex_c,1 ); /* 相互排除用のセマフォの初期化 */
sem_init( &rem,BUFFSIZE ); /* バッファの空きは、Ｎ，e=0 */
sem_init( &count,0 ); /* バッファは空，f=0 */

pthread_create(&thread_consumer_A, NULL, &consumer_A, NULL);
pthread_create(&thread_consumer_B, NULL, &consumer_B, NULL);
pthread_create(&thread_producer_A, NULL, &producer_A, NULL);
pthread_create(&thread_producer_B, NULL, &producer_B, NULL);
pthread_join(thread_consumer_A, NULL);
pthread_join(thread_consumer_B, NULL);
pthread_join(thread_producer_A, NULL);
pthread_join(thread_producer_B, NULL);
}

void *
producer_A(void *arg) /* 生産者プロセス */
{
int x ;
int i ;
x = 0 ;
fprintf(stderr,"producer_A(): started.¥n");
for( i=0 ; i<nloop ; i++ )
{
省略
fprintf(stderr,"producer_A(): put %d.¥n", x );
省略
}
}


void *
producer_B(void *arg) /* 生産者プロセス */
{
int x ;
int i ;
x = 0 ;
fprintf(stderr,"producer_B(): started.¥n");
for( i=0 ; i<nloop ; i++ )
{
省略
fprintf(stderr,"producer_B(): put %d.¥n", x );
省略
}
}


void *
consumer_A(void *arg) /* 消費者プロセス */
{

省略
fprintf(stderr,"consumer_A(): started.¥n");
for( i=0 ; i<nloop; i++){
sem_p( &count);
sem_p(&mutex_c);

x = buffer[j_c] ;
fprintf(stderr,"consumer_A(): get %d.¥n",x );
省略
}
```

```
}


void *
consumer_B(void *arg) /* 消費者プロセス */
{

省略
fprintf(stderr,"consumer_B(): started.¥n");
for( i=0 ; i<nloop; i++){
sem_p( &count);
sem_p(&mutex_c);

x = buffer[j_c] ;
fprintf(stderr,"consumer_B(): get %d.¥n",x );
省略
}

}
```

・実行結果

```
[j05002@6-synch]% ./mp-sem-prodcons2-2
consumer_A(): started.
consumer_B(): started.
producer_A(): started.
producer_A(): put 0.
producer_A(): put 2.
producer_A(): put 4.
producer_B(): started.
consumer_A(): get 0.
consumer_A(): get 2.
consumer_A(): get 4.
producer_A(): put 6.
producer_A(): put 8.
producer_A(): put 10.
consumer_A(): get 6.
consumer_A(): get 8.
consumer_A(): get 10.
producer_A(): put 12.
producer_A(): put 14.
producer_A(): put 16.
consumer_A(): get 12.
consumer_A(): get 14.
consumer_A(): get 16.
producer_A(): put 18.
producer_B(): put 0.
producer_B(): put 2.
consumer_A(): get 18.
consumer_B(): get 0.
consumer_B(): get 2.
producer_B(): put 4.
producer_B(): put 6.
producer_B(): put 8.
consumer_B(): get 4.
consumer_B(): get 6.
consumer_B(): get 8.
producer_B(): put 10.
producer_B(): put 12.
producer_B(): put 14.
consumer_B(): get 10.
consumer_B(): get 12.
```

```
consumer_B(): get 14.
producer_B(): put 16.
producer_B(): put 18.
consumer_B(): get 16.
consumer_B(): get 18.
[j05002@6-synch]%
```

(2)生産者の数が 2 で、消費者の数が 3

・ソース

ソースの場所：/home/y05/j05002/6-synch/mp-sem-prodcons3-2.c

(1)の場合に消費者を 1 つ加えただけなので、
ソースは省略する。

・実行結果

```
[j05002@6-synch]% ./mp-sem-prodcons3-2
consumer_A(): started.
consumer_B(): started.
consumer_B(): started.
producer_A(): started.
producer_A(): put 0.
producer_A(): put 2.
producer_A(): put 4.
producer_B(): started.
consumer_A(): get 0.
consumer_A(): get 2.
consumer_A(): get 4.
producer_A(): put 6.
producer_A(): put 8.
producer_A(): put 10.
consumer_A(): get 6.
consumer_A(): get 8.
consumer_A(): get 10.
producer_A(): put 12.
producer_A(): put 14.
producer_A(): put 16.
consumer_A(): get 12.
consumer_A(): get 14.
consumer_A(): get 16.
producer_A(): put 18.
producer_B(): put 0.
producer_B(): put 2.
consumer_A(): get 18.
consumer_B(): get 0.
consumer_B(): get 2.
producer_B(): put 4.
producer_B(): put 6.
producer_B(): put 8.
consumer_B(): get 4.
consumer_B(): get 6.
consumer_B(): get 8.
producer_B(): put 10.
producer_B(): put 12.
producer_B(): put 14.
consumer_B(): get 10.
consumer_B(): get 12.
consumer_B(): get 14.
producer_B(): put 16.
producer_B(): put 18.
consumer_B(): get 16.
```

```
consumer_B(): get 18.
^C
[j05002@6-synch]%
```

生産者の数も消費者の数も 2 の場合はバランスがよく、
うまく実行できているが、生産者の数が 2 で消費者の数
が 3 の場合は、バランスが悪く、デッドロックが起こっている。
消費者が情報が入るのを待っているが、生産を行うものがいない
ので、デッドロックが起こったと考えられる。