オペレーティングシステム
~ 相互排除と同期（2）~

山田 浩史
hiroshiy @ cc.tuat.ac.jp
2015/06/05

ロック (Lock)

- 排他制御のためのプリミティブ
  - 共有変数として Lock 型の変数を用意
    - lock とはカギのこと
  - Lock 型の変数に対する操作
    - Lock(Lock lock);
      - lock を獲得する
      - あるプロセスが lock を獲得すると，他のプロセスがその lock を獲得しようとしても Blocked 状態になる
    - Unlock(Lock lock);
      - lock を開放する
      - lock を獲得しようとしてBlocked 状態にあったプロセスがあれば，lock を獲得させ，処理を再開する

ロックによる排他制御

- 危険区域に入るときに Lock() を呼ぶ
  - 危険区域に鍵をかけるイメージ
- 出るときに Unlock() を呼ぶ
  - 鍵を開けるイメージ

```
Lock lock;

void Ti()
{
   for (;;) {
      Non_Critical_Section;
      Lock(lock);
      Critical_Section;
      Unlock(lock);
   }
}
```

危険区域という部屋に鍵をかける
- 鍵がかかっていたら，他のプロセスは入れない

鍵をあける
- 鍵があくのを待っていたプロセスは，危険領域に入れるようになる

AGENDA

- 排他制御用のプリミティブ
  - セマフォ (Semaphore)
- セマフォを使った相互排除問題の解法
  - 有限長バッファ問題 (Bounded-buffer Problem)
- デッドロック (Deadlock)
  - 排他制御用プリミティブを正しく使わないと生じる
- 高度な排他制御用プリミティブ
  - 条件変数 (Conditional Variable)
  - モニタ (Monitor)

セマフォ (Semaphore)

- より強力なプロセス間同期のための機構
  - 整数型の変数 S
  - S に対する操作 (いずれも不可分)
    - Wait(S) {
      if (S > 0) then S = S – 1;
      else スレッドの実行を一時停止する; }

    - Signal(S) {
      if (Wait(S)で停止しているスレッドがある)
      then 停止しているスレッドのひとつを再開する;
      else S = S + 1;
  - S の直感的な意味
    - S の値は Wait() を通過できるプロセス数を表す

セマフォによる排他制御

```
Semaphore S = 1;


void Ti()
{
   for (;;) {
      Non_Critical_Section;
      Wait(S);
      Critical_Section;
      Signal(S);
   }
}
```

Wait(S):
if S > 0 then S := S – 1
else suspend on the semaphore S;

Signal(S):
if there are processes suspended on S
then wake up one of them
else S := S + 1;

Wait(S); ← S が 1 なら, S を 0 にして危険区域に入る
S が 0 なら, S が 1 になるまで待つ

Signal(S); ← S を 1 にする

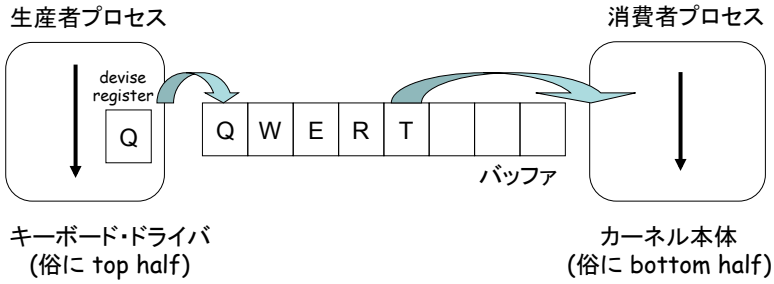
セマフォが効果的な場面

- 生産者･消費者問題
  - 生産者 (Producer)
    - 消費者が利用するデータを生成するプロセス
  - 消費者 (Consumer)
    - 生産者が生成したデータを利用するプロセス
  - 生産者と消費者で同期をとりながら処理を行う
    - 消費者は生産者がデータを生成するまで待つ
    - 生産者は消費者がデータを利用しおわるまで待つ

  例: プリンタ･スプーラ，キーボードドライバ etc.

Bounded Buffer Problem

- 生産者と消費者が有限長のバッファを用いて
  データをやりとりする問題
  - 単純な相互排除問題よりも難しい
  - Lock だけで実現するのは大変
    - 不可能ではない
- OS の内部では多くの bounded buffer が
  利用されている
  - キーボードドライバ，ネットワーク，ウィンドウシステムなど



セマフォによる解法

```
int B[N];
int In_Ptr = 0, Out_Ptr = 0;
Lock lock;
Semaphore Elements = 0;
Semaphore Spaces = N;

void Producer()
{
   int data;
   for (;;) {
      data = Produce();
      Wait(Spaces);
      Lock(&lock);
      B[In_Ptr] = data;
      In_Ptr = (In_Ptr + 1) % N;
      Unlock(&lock);
      Signal(Elements);
   }
}

void Consumer()
{
   int data;
   for (;;) {
      Wait(Elements);
      lock(&lock);
      data = B[Out_Ptr];
      Out_Ptr = (Out_Ptr + 1) % N;
      Unlock(&lock);
      Signal(Spaces);
      Consume(data);
   }
}
```

解法の心

- Elements
  - バッファに入っている要素の数を表す
- Spaces
  - バッファの空きスロットの数を表す

  ※ Elements + Spaces = N

- Producer の Wait (Spaces)
  - バッファ内の空きスロットが 0 なら待つ
- Consumer の Wait(Elements)
  - バッファ内の要素の数が 0 なら待つ
- Producer の Signal(Elements)
  - 要素を待っている Consumer がいれば，Consumer を再開する
- Consumer の Signal(Spaces)
  - 要素を入れるためのスロットを待っている Producer がいれば，Producer を再開する

排他制御による不具合

- デッドロック (Deadlock)
  - 2 つ以上のスレッドが互いに資源の解法を待ち合って処理が先に進まなくなること
  - 排他制御用プリミティブを正確に使わないがために起こる現象

Deadlock が起きる例

```
void () thread1()
{
    Lock(A); // 1. 実行後 thread2へ
    Lock(B); // 4. ここでサスペンド
    何か処理を行う
    Unlock(B);
    Unlock(A);
}
```

```
void () thread2()
{
    Lock(B); // 2. 実行
    Lock(A); // 3. ここでサスペンド
    何か処理を行う
    Unlock(A);
    Unlock(B);
}
```

デッドロックの回避方法の 1 例

- ロックを獲得する順番を統一する

例: ロック A, B の順番でロックを獲得するようにする

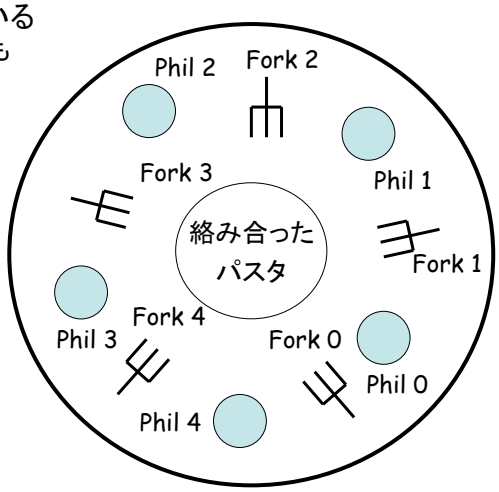
```
void () thread1()
{
    Lock(A);
    Lock(B);
    何か処理を行う
    Unlock(B);
    Unlock(A);
}
```

```
void () thread2()
{
    Lock(A);
    Lock(B);
    何か処理を行う
    Unlock(B);
    Unlock(A);
}
```

- 排他制御を行うときは細心の注意を払うこと！
  - 排他制御用プリミティブの誤用による他の不具合
    - ライブロック(Livelock) など

食事をする哲学者問題 (1/2)

- Dining Philosopher Problem
- 5 人の哲学者が食事をしている
  - 哲学者なので食事しながらも考える
  - 食事をするには
    - 2 つのフォークでテーブルの中央にあるパスタを食べる
    - パスタは絡まっていてフォークが 2 つ必要
  - 食べないと死ぬ
- 5 人の哲学者を死なせずに食事をさせるには?



食事をする哲学者問題 (2/2)

- 条件
  - フォークを 2 つ持たないと食事ができない
  - 哲学者は自分の両脇のフォークしか使えない
  - 2 人の哲学者は同時に同じフォークを使えない
- 目標
  - 誰もが食事できない状態になってはいけない
  - 誰か一人が長い間食事のできない状態になってはいけない

First Attempt

- 哲学者をプロセスとして表す
- フォークをセマフォとして表す
  - 自分の右側のフォーク、次に左側のフォークを取る

```
Semaphore Fork[] = {1, 1, 1, 1, 1};

void Philosopher_I()
{
   int I = 哲学者の番号 (0 から 4);
   for (;;) {
      Think();
      Wait(Fork(I));
      Wait(Fork((I + 1) % 5));
      Eat();
      Signal(Fork(I));
      Signal(Fork((I + 1) % 5));
   }
}
```

First Attemp は失敗

- デッドロックが生じてしまう
  - 哲学者全員が同時に Wait(Fork(I)); を実行すると・・・
    - Wait(Fork((I + 1) % 5)); で永久に待ってしまう

  ⇒ 哲学者はみんな餓死してしまう

Second Attempt

- 哲学者 4 だけ別行動をとる
  - 0 番から 3 番の哲学者は自分の右側からフォークをとる
  - 4 番の哲学者は自分の左側のフォークからとる

```
void Philosopher_4()
{
   int I = 4;
   for (;;) {
      Think();
      Wait(Fork(0));
      Wait(Fork(4);
      Eat();
      Signal(Fork(4));
      Signal(Fork(0);
   }
}
```

- これはうまく動作する
  - 4 番目の哲学者だけ有利/不利にならないの？
- 他の解法
  - 1 度に 4 人だけが食事をする
  - 2 つ同時にフォークが掴めなかったらフォークを離す

などなど

セマフォは強力なんだけど･･･

- 同期の記述が直感的でない場合が多い
  - セマフォの wait()
    - if (S > 0) then S = S – 1; else スレッドを一時停止;
  - S の値が 0 より大きいかどうかだけが判断基準
    - 同期の条件を S > 0 となるように記述を工夫しなくてはならない
    - Bounded Buffer Problem で記述したい条件:
      - バッファに空きがあるかないか
      - バッファにデータがあるかどうか
    - セマフォを用いた解法では S の値を巧妙に操作した
      - spaces = N, elements = 0

高レベルな排他制御用プリミティブ

- 条件変数 (Condition Variable)
  - 排他制御および同期のためのプリミティブ
  - POSIX Thread ライブラリで採用
- モニタ (Monitor)
  - 排他制御および同期のためのプログラミング言語機能
  - Java のスレッド機構で採用

条件変数(Condition Variable)

- ロック変数とペアで用いる変数
  - 条件待ちを表すための変数
  - セマフォより複雑な条件が簡単に記述できる
- 条件変数に対する操作
  - 変数: ConditionVariable c;
  - 変数 c に対する操作
    - wait(ConditionVariable *c, Lock *lock);
      - 条件待ちのための操作
      - lock を解放する
      - 条件変数 c に結びつけられたキュー上で待つ
      - シグナルを受け取ると lock を確保してから起き上がる
    - signal(ConditionVariable *c)
      - 通知のための操作
      - 条件変数 c に結びつけられたキュー上で待っているプロセスのひとつを起こす

条件変数による解法

```
int B[N];
int In_Ptr = 0, Out_Ptr = 0;
Lock lock;
ConditionVariable full, empty;
int count = 0;

void Producer()
{
   int data;
   for (;;) {
      data = Produce();
      Lock(lock);
      while (count >= N) Wait(full, lock);
      count = count + 1;
      unlock(lock);
      B[In_Ptr] = data;
      In_Ptr = (In_Ptr + 1) % N;
      Signal(empty);
   }
}
```

```
void Consumer()
{
   int data;
   for (;;) {
      Lock(lock);
      while (count ==  0) Wait(empty, lock);
      count = count – 1;
      unlock(lock);
      I = B[Out_Ptr];
      Out_Ptr = (Out_Ptr + 1) % N;
      Signal(full);
      Consume(data);
   }
}
```

モニタ (Monitor)

- Lock, Unlock を言語機構に組み入れたもの
  - モニタは共有データ，共有データに
    アクセスするための手続き群を持つ
  - モニタの外から共有データに直接アクセスできない
    - アクセスしようとするとコンパイル時にエラーとして検出される
  - モニタ内で実行中のプロセスは高々ひとつに制限される
    - すでに他のプロセスが実行中の場合，待たされる

```
monitor sample {
   int shared_variables;
   void hogehoge() { /* 共有変数にアクセスする */ }
   void pogepoge() { /* 共有変数にアクセスする */ }
}
```

モニタによる解法

```
monitor Producer_Consumer_Monitor {
   int B[N];
   int In_Ptr = 0, Out_Ptr = 0;
   int Count = 0;
   ConditionVariable Full, Empty;

   void Producer()  {
      int data;
      data = Produce();
     if (Count == N) Wait(Full);
      B[In_Ptr] = data; In_Ptr = (In_Ptr + 1) % N; Count = Count + 1;
      Signal(Empty);
   }

   void Consumer() {
      int data;
      if (Count == 0) Wait(Empty);
      data = B[Out_Ptr]; Out_Ptr = (Out_Ptr + 1) % N; Count = Count – 1;
      Signal(Full);
   }
}
```

まとめ

- 相互排除と同期について学んだ
  - セマフォ
  - 生産者･消費者問題
  - デッドロック
  - 条件変数
  - モニタ