8. 共有記憶を利用したプログラミング

プロセスとスレッド

Pスレッド

並列プログラミング言語の特徴

8.1 共有記憶マルチプロセッサ(1)

共有記憶マルチプロセッサシステム

- 全てのプロセッサが同一のアドレス空間を持つ

単一バスを利用した共有メモリマルチプロセッサ

- 全てのプロセッサが同一の信号線の組(バス)に接続
- バスは一度に一つのプロセッサしか使用できない
- プロセッサが少数の時に有効

複雑な接続ネットワークを利用した共有メモリマルチプロセッサ

- 同時に複数のプロセッサメモリにアクセス可能 -> 例えばオメガ網
- ネットワークが高価になりやすい

プロセッサにキャッシュを持たせた共有メモリマルチプロセッサ

- キャッシュのコントロールはやや複雑

8.1 共有記憶マルチプロセッサ(2)

共有メモリ型並列処理のためのプログラミング

- 新しいプログラミング言語を使用する
- 既存の逐次言語を修正する
- 既存の逐次言語とライブラリ関数を使用する
- 逐次言語を利用し並列化コンパイラで並列化する

並列プログラミング言語

- Concurrent Pascal, Pascalの拡張
- Ada, 米国国防局が開発した新しい言語
- Concurrent C, Cの拡張版
- 残念ながらどれも普及していない

スレッド

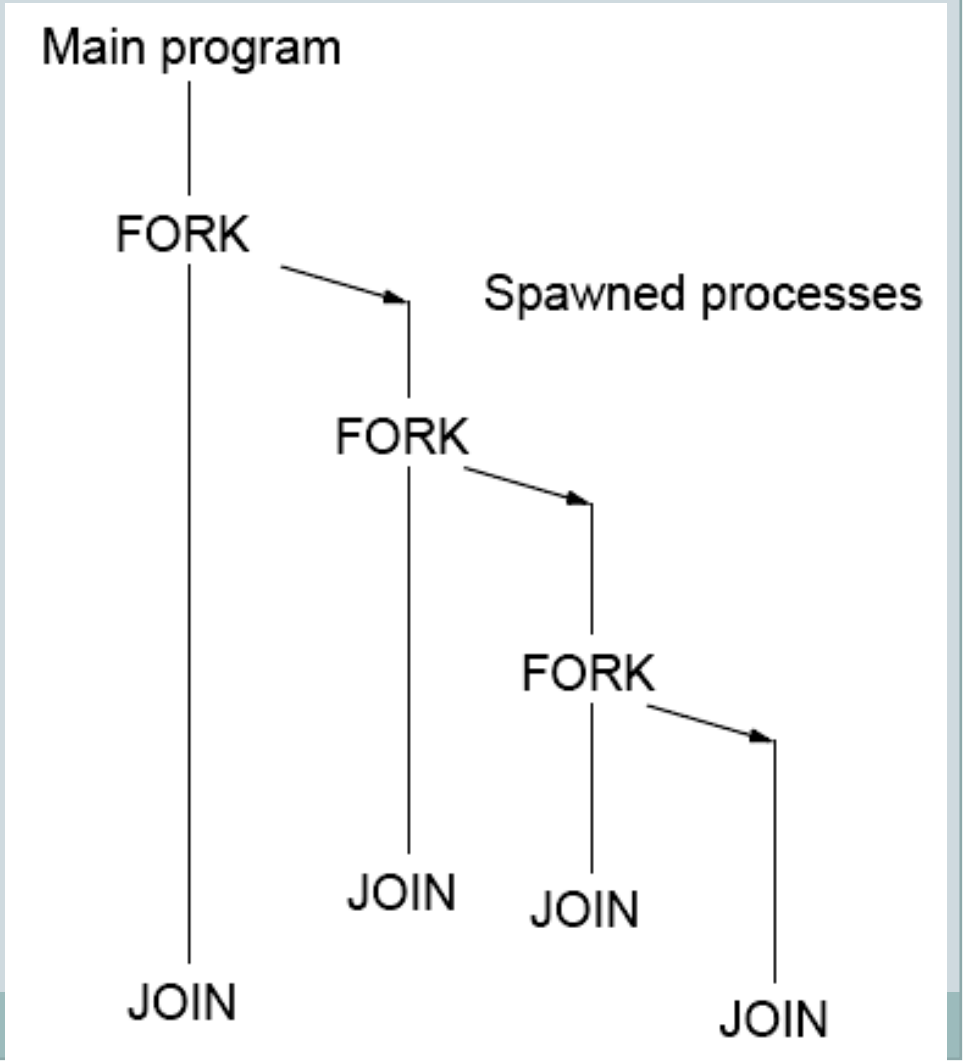
- IEEE標準スレッドのPOSIXスレッド(Pスレッド)
- Javaもスレッド機能を備えている

8.2 並列性を記述するための構文(1)

並列プロセスの生成

FORK-JOIN

- FORKで並列プロセスを生成し
- JOINで結合し逐次になる
- 多くの並列プロセス生成のためには入れ子構造が必要



8.2 並列性を記述するための構文(2)

UNIX

- 単一プロセッサのUNIXではTSSでプロセスを切り替えながら処理を行う
- 複数プロセッサであれば並列処理が可能
- プロセス生成システムコール等を使えば並列プログラムが書ける

fork()

- 新しいプロセス(子プロセス)を生成する
- 子プロセスは、独自のプロセスIDを持つ
- 子プロセスの中身は呼び出したプロセスのコピーである
- 成功すれば子プロセスに0を返し、親プロセスに子プロセスのIDを返す

8.2 並列性を記述するための構文(3)

wait(startup)

- シグナルを受け取るまで，あるいは子プロセスのうちの一つが終了するか中断する場で，呼び出しプロセスを待たせる

exit(status)

- プロセスを終了する

FORK-JOINの記述

```
pid=fork();   /* fork */
  …
  …
```

子プロセスと親プロセスの両方が実行するプログラムコード

```
  …
  …
if(pid==0) exit(0); else wait(0); /* join */
```

8.2 並列性を記述するための構文(4)

スレッド

UNIXの通常のプロセスは重量プロセス

- プロセス毎に変数，スタック，メモリ割当てがある
- プロセスの生成は，親プロセスの完全なコピーで，プログラムコード，ヒープ，スタックが新たに生成される
- 利用されない部分が多い(fork()以前でのみ使われる変数，コード)

軽量プロセス(スレッド)

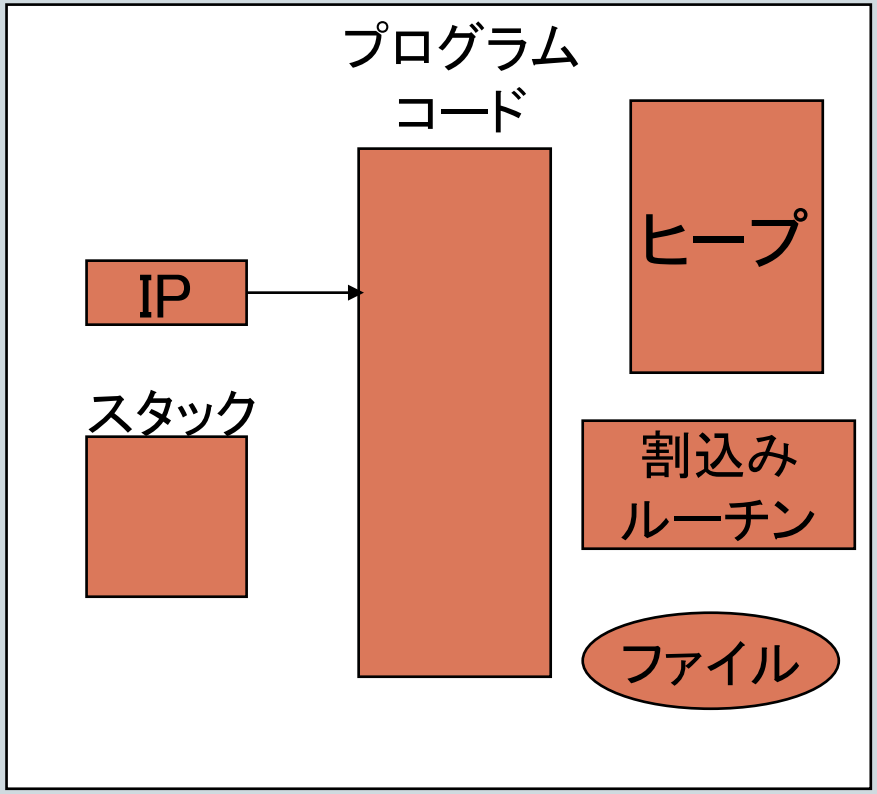
- 複数のスレッドがメモリ空間と大域変数を共有する
- スタックは異なる

プロセスとスレッドの違い

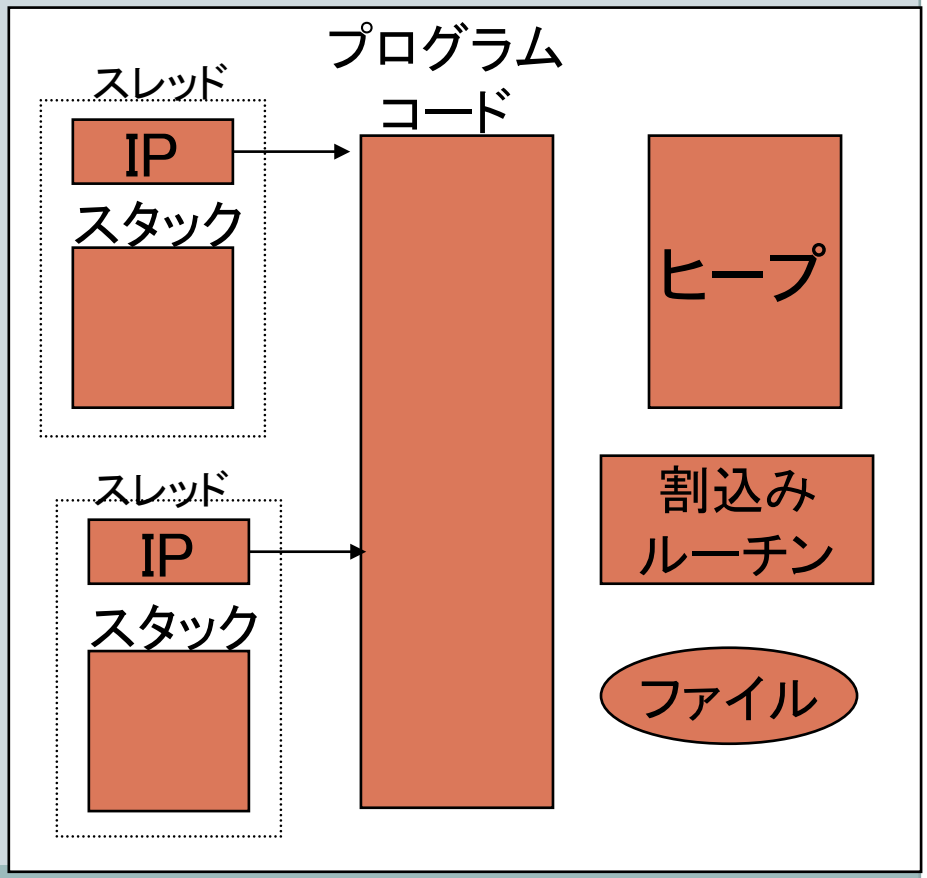
- スレッドの生成は、プロセスの生成に比べて約3桁少ない時間
- スレッド間で共有帯域変数を持つ
- スレッド間同期はプロセス間同期よりも容易

8.2 並列性を記述するための構文(5)

プロセスの構成



(マルチ)スレッドの構成



8.2 並列性を記述するための構文(6)

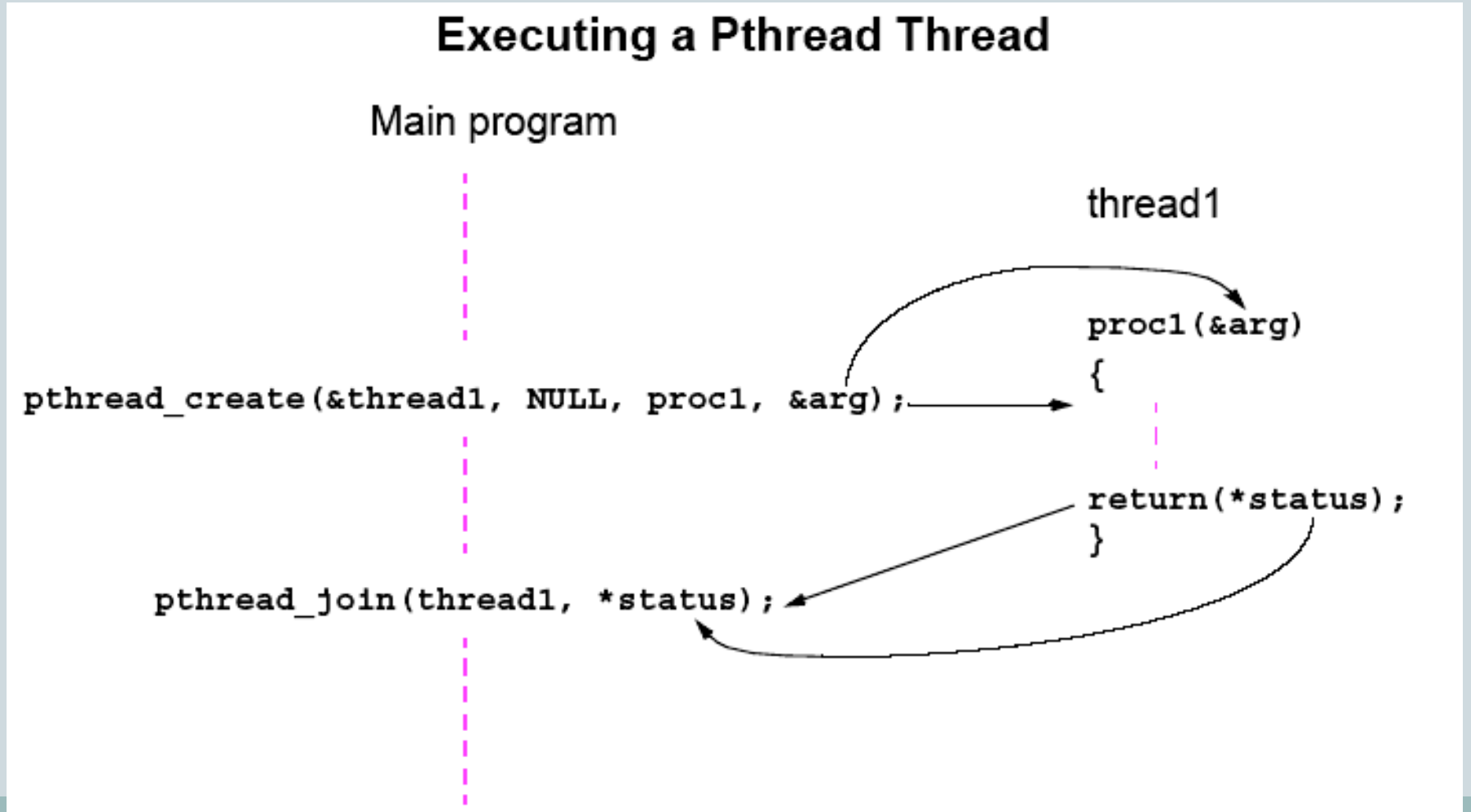
Pスレッド(IEEE Portable Operating System Interface POSIX)

- 標準的なスレッド
- 多くのプラットフォームで動作可能

スレッドの実行

- PスレッドによるFORK-JOINの実行
  - pthread_create(..), ptherad_join(..)を使って記述できる

8.2 並列性を記述するための構文(7)



8.2 並列性を記述するための構文(8)

例:

```
1.  main(){
2.      ...
3.      pthread_t thread1; /* Pスレッドのデータ型を扱う */
4.      ...
5.      ...
6.      pthread_create(&thread1, NULL, proc1, &arg);
7.      ...
8.      pthread_join(thread1, *status);
9.      ...
10. }
11. proc1(arg)
12. {
13.    ...
14.    ...
15.    return(*status);
16. }
```

8.2 並列性を記述するための構文(9)

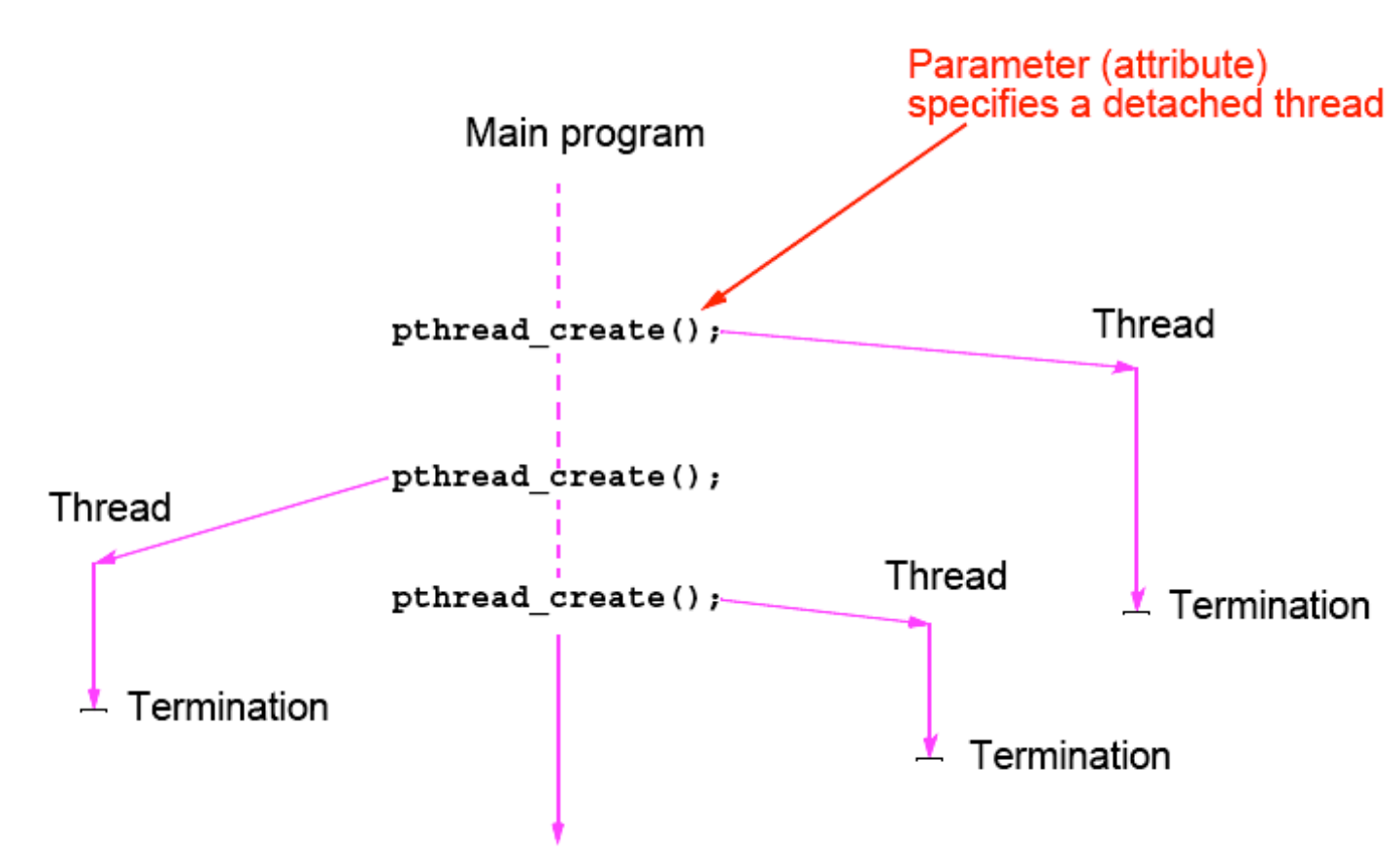
バリアの例:

```
1.   ...
2.   for(i=0; i<n; i++)
3.     pthread_create(&thread[i], NULL, (void *)slave,
     (void *)&arg);
4.   ...
5.   ...
6.   for(i=0; i<n; i++)
7.     pthread_join(&thread[i], NULL);
8.   ...
9.   ...
```

8.2 並列性を記述するための構文(10)

切り離されたスレッド

- joinを必要としないスレッドは切り離した方が効率的である



8.3 共有データ(1)

複数のプロセスからアクセスできる変数やデータ構造を共有記憶につくる

共有データの作成

- UNIXのプロセス(重量プロセス)がデータ共有するには
  - 共有記憶のためのシステムコールが必要 shmget(), shmat()
- スレッドがデータ共有するには
  - メインプログラムの先頭で宣言された変数はグローバルで全てのスレッドからアクセス可能

共有データへのアクセス

- 二つのプロセスが共有変数xに対して、x=x+1 を実行する場合を考える。

```
1.  process1                    process2
2.    read x                      read x
3.    calculate x+1             calculate x+1
4.    save x                      save x
```

どうなる？

8.3 共有データ(2)

共有データアクセスに関する問題の一般化

- 共有資源があり同時にアクセスできるプロセス数に制限がある
- 例えば，共有資源がひとつあり，同時に唯一つのプロセスのみアクセス可

クリティカルセクション

- 一度に唯一つのプロセスのみしか資源をアクセスできないようなプログラム領域

相互排除(Mutual Exclusion)

- クリティカルセクションを正しく実行するようなメカニズム
- 同時には唯一つのプロセスのみクリティカルセクションを実行可能
- クリティカルセクションの実行を終えたら他のプロセスが入ることを許す

ロック(Lock)

- 最も簡単な相互排除の機構
- ロックは1ビット変数：1はあるプロセスがクリティカルセクションに入っている
- 0はどのプロセスも入っていない

8.3 共有データ(3)

ロックの例

```
1.  while(lock==1) do_nothing; /* while loopでlockが0になるのを待つ*/
2.  lock=1;
3.     ....
4.     ....
5.     critical section
6.     ....
7.  lock=0;                    /* leave the critical section */
```

busy waiting

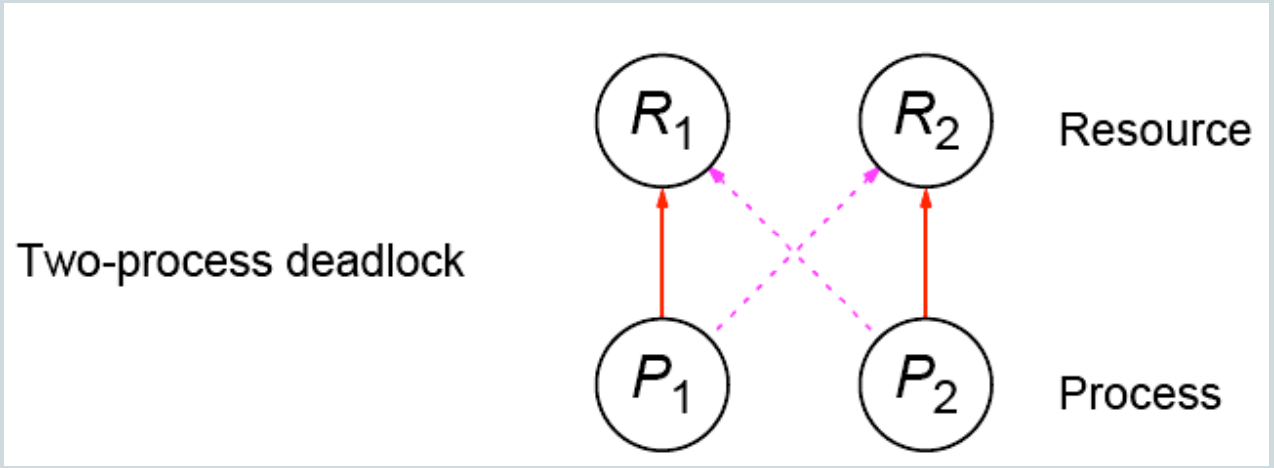
- 無駄にループをまわっている
- プロセッサ利用の効率が悪い

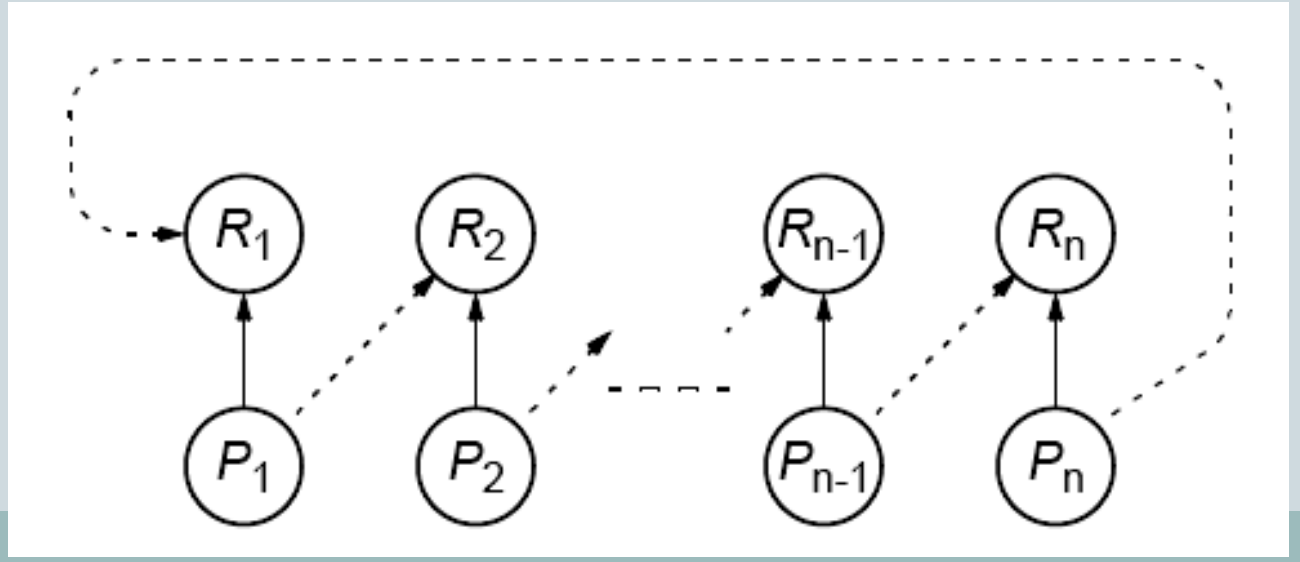
8.3 共有データ(4)

Pスレッドロックルーチン

```
1.  pthread_mutex_t mutex1;
2.  ...
3.  ...
4.  pthread_mutex_init(&mutex1, NULL);
5.  ...
6.  ...
7.  pthread_mutex_lock(&mutex1);
8.   critical section
9.  pthread_mutex_unlock(&mutex1);
```

8.3 共有データ(5)　デッドロック





8.3 共有データ(6)

セマフォ

- Dijkstraが1968に提案
- P操作とV操作

P操作 P(s):

- s>0ならs=s−1とし，プロセスに処理の継続を許可する
- s=0なら待ち状態になる．ただし，V(s)からシグナルが届いたら，次の処理を続行する

V操作 V(s):

- 待っているプロセスがいればそのうちの一つにシグナルを送る
- 待っているプロセスがなければ，s=s+1とする

8.3 共有データ(7)

バイナリセマフォの例

Process 1	Process 2	Process 3
Noncritical section	Noncritical section	Noncritical section
.	.	.
.	.	.
.	.	.
P(s)	P(s)	P(s)
Critical section	Critical section	Critical section
V(s)	V(s)	V(s)
.	.	.
.	.	.
.	.	.
Noncritical section	Noncritical section	Noncritical section

8.3 共有データ(8)

モニタ(Monitor)(Hoare, 1974)

- セマフォは機能的には問題ないが，低レベル記述なので人間に分かりづらい
- プログラミング言語でいうアセンブラみたいなもの
- モニタは，より高級言語に近い
- 共有データとそれに対する操作群がカプセル化されている

動作

- モニタの操作群を通してのみ共有データにアクセス可能
- あるプロセスがアクセス中に別のプロセスがアクセス要求をすると，要求したプロセスは待ち行列にいれられる
- アクセス中のプロセスがモニタの終了を終えたら，待ち行列の最初のプロセスが手続きを使用することを許される

8.3 共有データ(9)

```
1.  monitor_proc1()
2.  {
3.      P(monitor_semaphore);
4.      ...
5.      ...
6.      ...
7.      V(monitor_semaphore);
8.      return;
9.  }
```

Javaにもモニタの機能がある

8.3.3 並列性のための言語構文(1)

共有データ

- 複数のプロセスから参照可能
- 通常相互排除までは保証しない
- shared int x;
- あるいは
- int * global x;

par構文

- 並列実行を明示する
- 命令レベルの並列化
- 一般に粒度が小さいので、多くのシステムではオーバーヘッドが無視できない

```
1.  par{
2.      S;
3.      S2;
4.      ...
5.      Sn;
6.    }
```

8.3.3 並列性のための言語構文(2)

粒度が大きい(例えばスレッド)場合の記述

```
1.    par{
2.        proc1();
3.        proc2();
4.        ...
5.        proc_n();
6.      }
```

forall構文

- 複数の同様な処理を並列に実行する
- 例えば, 同じ処理を異なる複数のデータに対して行う

```
1.    forall(i=0; i<n; i++){
2.        S1;
3.        S2;
4.        ...
5.        S_m;
6.      }
```

8.3.4 依存性の解析(1)

並列プログラミングの鍵

- 並列処理を行うために，処理の中の並列性を抽出する
- 並列性を抽出するためには，依存性を調べる

依存性解析

- プログラムの中の依存性を調べる
- 依存性のない処理の例：

```
1.   for (i=0; i<5; i++)
2.        a[i]=0;
```

は、a[1]=0, a[2]=0, …, a[4]=0の間に依存性がないので，

```
1.   forall (i=0; i<5; i++)
2.        a[i]=0;
```

と同じ意味を持つ。次はどうか？

```
1.   for(i=2; i<6; i++){
2.        x=i-2*i+i*i;
3.        a[i]=a[x];
4.     }
```

8.3.4 依存性の解析(2)

バーンスタインの条件(Bernstein's Conditions)

- 二つのプロセスが同時に実行できるかどうかを決定するための十分条件

I_i: プロセスP_iによって読み込まれるメモリロケーション(変数)
O_j: プロセスP_jによって変更されるメモリロケーション

バーンスタインの条件(Bernstein's Conditions)

- 同時に実行されるプロセスP_1とP_2に対して
- $I_1 \cap O_2 = \phi$; P_1への入力がP_2の出力の一部になっていない
- $I_2 \cap O_1 = \phi$; P_2への入力がP_1の出力の一部になっていない
- $O_1 \cap O_2 = \phi$; 二つのプロセスの出力が重なっていない

8.3.4 依存性の解析(3)

例:

1. P_1: a = x+y;
2. P_2: b = x+z;

の時,

- I_1={x,y}, I_2={x,z}, O_1={a}, O_2={b}

なので, バーンスタインの条件(Bernstein's Conditions)は成立している.

1. P_1: a = x+y;
2. P_2: b = a+b;

の時はどうか?