プロセス管理の後半で、並行プロセスに関してです。教科書は第8章の多重プロセスです。

# オペレーティングシステム
## (2024年 第9回)

## プロセス管理について(2)

# 前回の課題について

詳細は「第8回の課題について.pdf」をみてください

（1）FATのエントリで管理されるクラスタ番号が16ビットで表現されるので、ほぼ 64K（$2^{16}=65536$）個のクラスタを指定することができます
クラスタのサイズが32KBですので、64K個×32KB＝2048MB（2GB）となります

（2）データブロックのサイズが4KBで、p.21の図で示されているように直接ブロックを参照するエントリは12個あるので、直接の部分で構成できるファイルのサイズは　12×4KB＝48KB
一段間接では、間接ブロック（サイズは4KB）がポイントできるブロックは1024個（ブロックをポイントするエントリが4バイトであるとする）あるので、ここで構成できるファイルのサイズは　1024×4KB＝4MB
二段間接の場合は、1つの間接ブロックにさらに、それぞれもう1つ間接ブロックを設けるので、ここで構成できるファイルのサイズは　1024×1024×4KB＝4GB
三段間接では、さらにもう1段1024個の間接ブロックがあるので、ここで構成できるファイルのサイズは
1024×1024×1024×4KB＝4TB
となり、最大のサイズはこれらを加算したものになります

$$12\times4\mathrm{KB} + 1024\times4\mathrm{KB} + 1024\times1024\times4\mathrm{KB} + 1024\times1024\times1024\times4\mathrm{KB} \fallingdotseq 4\mathrm{TB}$$

# 並行プロセス

- プロセスは、複数が「同時に」実行される
  - 並行プロセス（concurrent process）
    見かけ上同時に複数動作しているように見えるプロセス
  - 並列プロセス（parallel process）
    実際に同時に実行されるプロセス
- 多くのアプリケーションは独立して動作し、相互干渉しないプロセスとして「同時に」実行される
  問題が発生することもある
  - 複数のアプリケーションで同じファイルへの書き込みなど共通の資源を利用しようとすると正しい結果が得られないことがある
  - 相互干渉の影響を回避する仕組みが必要
- プロセス間の相互干渉を積極的に利用する方式もある
  - 複数のプロセッサで計算処理を分割・集計して実行

マルチプロセス環境ではプロセスは複数が「同時」に実行されますが、見かけ上（実際は違う）同時に複数動作しているように見えるプロセスを並行プロセス、実際に同時に実行されるものを並列プロセスと呼びます。

多くのアプリケーションは基本的には互いに独立で、個別のプロセスとして実行されて相互干渉することはありません。
しかし、複数のアプリケーションが共通の資源（ファイルなど）を利用していて、同時に書き込みしようとした場合などは正しい結果が得られないことがあります。

このような相互干渉を避ける仕組みが必要です。

また、一方、プロセス間の相互干渉を積極的に利用して計算することもあります。たとえば、処理の高速化のために、マルチプロセッサの計算機で複数のプロセッサで処理を分割して計算し、集計するような場合です。

# プロセス間の相互作用と制御

- 競合 … 複数のプロセスが共用する資源を同時に使用しようとする場合に発生

  → 共用する資源を同時に使用させないよう 排他制御 を行う

  - 相互排除 … 1つのプロセスにしか使用を許さない（不可分な処理の提供）
  - クリティカルセクション … 共用資源にアクセスする部分
    相互排除の対象

- 協調 … 複数のプロセスが協力して共通の目的のために処理を実行する形態
  プロセスの実行は、関連する他のプロセスに依存する

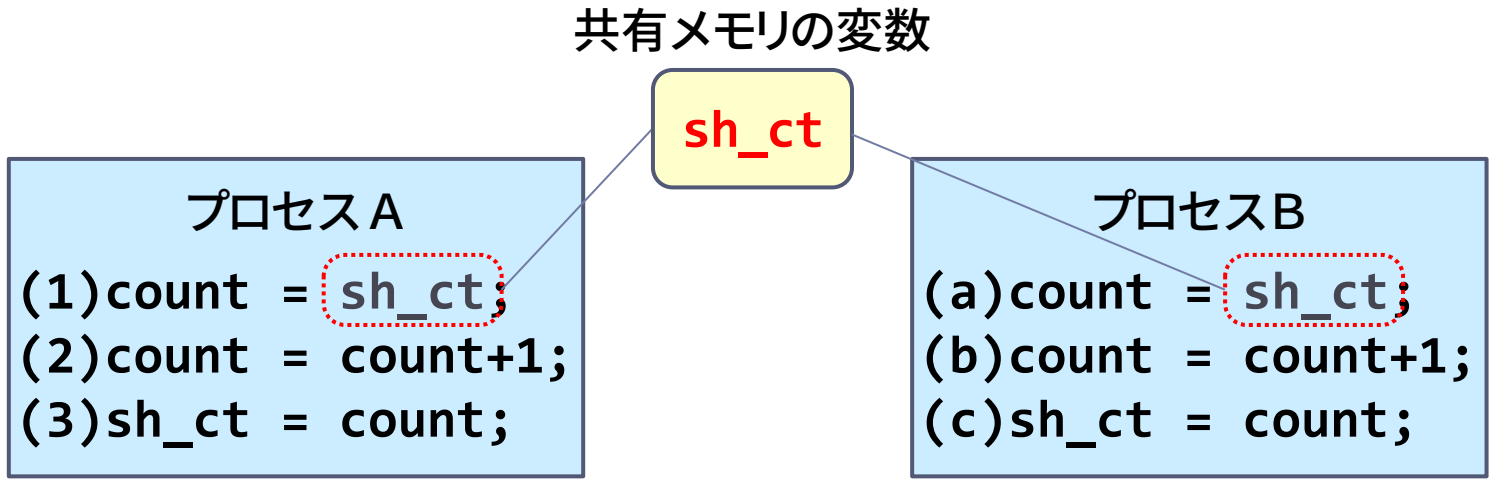
  → 共用する資源へのアクセスが適切に行われるかどうかをプロセス間で通知しあうため 同期 を行う

このようにプロセス間の相互作用には、同じ資源を同時に使用しようとして不具合が発生する「競合」と、協力して目的を達成するために依存関係を正しくする「協調」があります。

競合を避けるためには、1つだけがアクセスすることを保証する「排他制御」、協調のためにはアクセスする依存関係が適切であることを保証する「同期」が必要になります。

# 排他制御

- プロセス間共有資源の利用の問題点
  - 共有データに対する演算




プロセスAとBが同時に実行された場合、正しい結果を返さない

- A、Bのプロセス切り替えはどこで発生するか分からない。
  (2)の直後にBに切り替わって(a)(b)(c)と実行し、Aに戻って(3)が実行される など

まず、排他制御についてです。

共有メモリにあって、2つのプロセスで共有されるデータに対する演算について考えます。プロセスAとBでそれぞれ共有変数sh_ctを読んで、その値に1を加え、結果を再びsh_ctに返すというコードを考えます。仮にsh_ctが1で始まり、A→Bと実行されると、sh_ctは3になります。

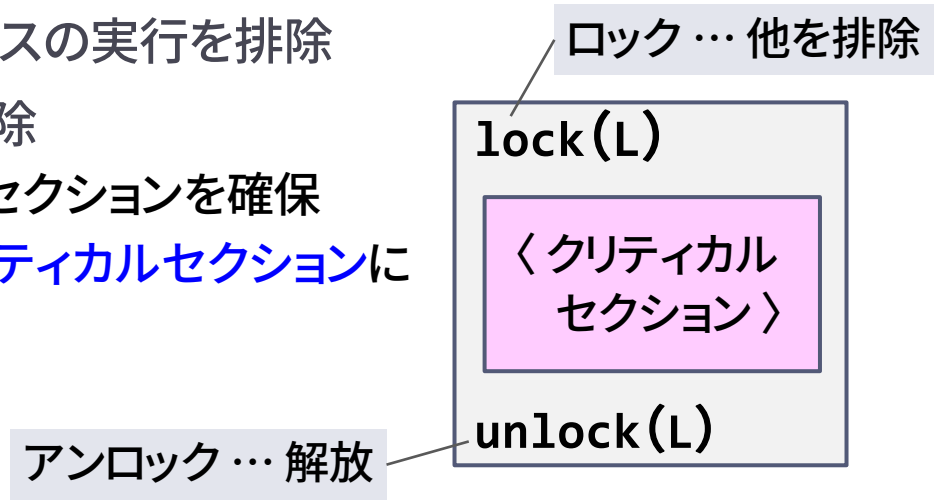
AとBが同時に実行されるとどうなるかということですが、A、Bのプロセス切り替えがどこで起こるかによります。

(2)の直後(sh_ctは1、Aのcountは2です)でBに切り替わると、sh_ctは1ですので、(b)でBのcountは2となり、(c)でsh_ctが2となります。

その直後にAに切り替わると、(3)が実行され、sh_ctはAでのcountの2が代入され、2のままです。(A、Bが逐次に実行された場合と結果が異なります)

# 排他制御の必要性

- 一つのプロセスでは正しく動作した処理も、共有資源がある場合は、複数のプロセスで同時に実行されると正しい動作をしないことがある
- 前記(1)～(3)、(a)～(c)が同時に実行されないようにすれば、この問題は解決する
- 排他制御によって、(1)～(3)、(a)～(c)の処理をそれぞれ一つの単位としてまとめ、ただ一つのプロセスが実行されるようにする
  - 割り込み禁止 … 他のプロセスの実行を排除
  - ビジーウェイトによる相互排除
    - ロック変数でクリティカルセクションを確保
    - ビジーウェイトによってクリティカルセクションに入れるまで待つ
  - セマフォによる制御




このように共有資源がある場合に、複数のプロセスで同時に実行すると動作が正しくなくなることがあります。

前の例では(1)～(3)と(a)～(b)が同時に実行されないようにすれば、問題が解決します。

このように、同時に実行されることがないようにすることを排他制御と呼びます。他のプロセスから実行されては困る区間(クリティカルセクション)を構成し、そこでは他のプロセスが実行されないようにします。

割り込みを禁止するだけでは不十分で、変数(ロック変数と呼ばれます)をループによって検査し、ロックできればクリティカルセクションを実行するもの(第3回に「busy wait」として出てきました)、セマフォと呼ぶ方式などがあります。

# 排他制御の実現(1)

- ビジーウェイトによる相互排除

```
while (flag == 1);
flag = 1;
〈クリティカルセクション〉
flag = 0;
```

lock()

unlock()

共有メモリの変数

- この一連の処理(lock()の開始からunlock()の終了まで)は割込み禁止で実行
- マルチプロセッサシステムではうまく動作しない

- TAS (Test And Set) 命令

```
if (flag == 1)
  return 1;
else {
  flag = 1;
  return 0;
}
```

この動作中は、他のプロセッサによるflagへのアクセスが禁止される(すなわち、不可分の動作として実行される)プロセッサ命令

これは1つの資源に対してロックを確保できたか否かという2値を扱うもの
一般的な問題として、その制限を取り払って考える必要がある
→ 代表的な排他制御法が セマフォ (Semaphore)

ビジーウェイトによる相互排除の実現はこのようなイメージです。共有メモリの変数flagを設け、これが0か1をとるようにします。
flagが1であると、だれかがすでに他を排除して処理しているので、自分はクリティカルセクションに入れず、このwhile文でループして待つことになります。

他での排除が終了するとflagが0になるので、この処理はクリティカルセクションに入れます。そこで、flagを1としてロックをかけ、他を排除します。自分の処理が終わればflagを0としてアンロックし、排除を解除します。

この一連の処理を割込み禁止で行えばうまくいきそうですが、マルチプロセッサでは、他のCPUから処理されることを排除できずうまく動作しません。
これは、flagを判定して1でなければ1を設定するというところ(その途中)で、他のプロセッサでも1を設定することを避けられないからです。

そこで、途中で他のCPUから実行できないように、特別の命令「flagを判定して1でなければ1を設定する」を他のCPUから操作されず、不可分の動作として実現するものを用意します。TAS(Test And Set)命令などと呼ばれるものです。

ここで説明したものは、1つの資源に対してロックを確保できたかどうかというものですが、2つ以上に一般化して考えるものが次のセマフォです。

# 排他制御の実現(2)

- セマフォによる排他制御
  - 同期用の整数変数：セマフォ sem（初期値は資源の数）
  - セマフォを引数とする操作、PとV
    lock、unlock の機能を含んだもの
  - P(sem) により、
    - sem が1以上なら、sem = sem − 1として処理を終了
    - それ以外は、実行したプロセスを待ち状態に → セマフォ に対応したキューに入れ、スケジューラに制御を渡す
  - V(sem) により、sem = sem + 1とし、
    - セマフォに対応したキューにプロセスがあれば、それを外して実行可能状態とし、スケジューラに制御を渡す
    - キューにプロセスがなければ、処理を完了してV(sem)を呼び出したプロセスを続行
  - sem の操作は不可分でなければならない
  - lock/unlock は semの初期値=1 の場合

ある資源がいくつ使用可能かを表す

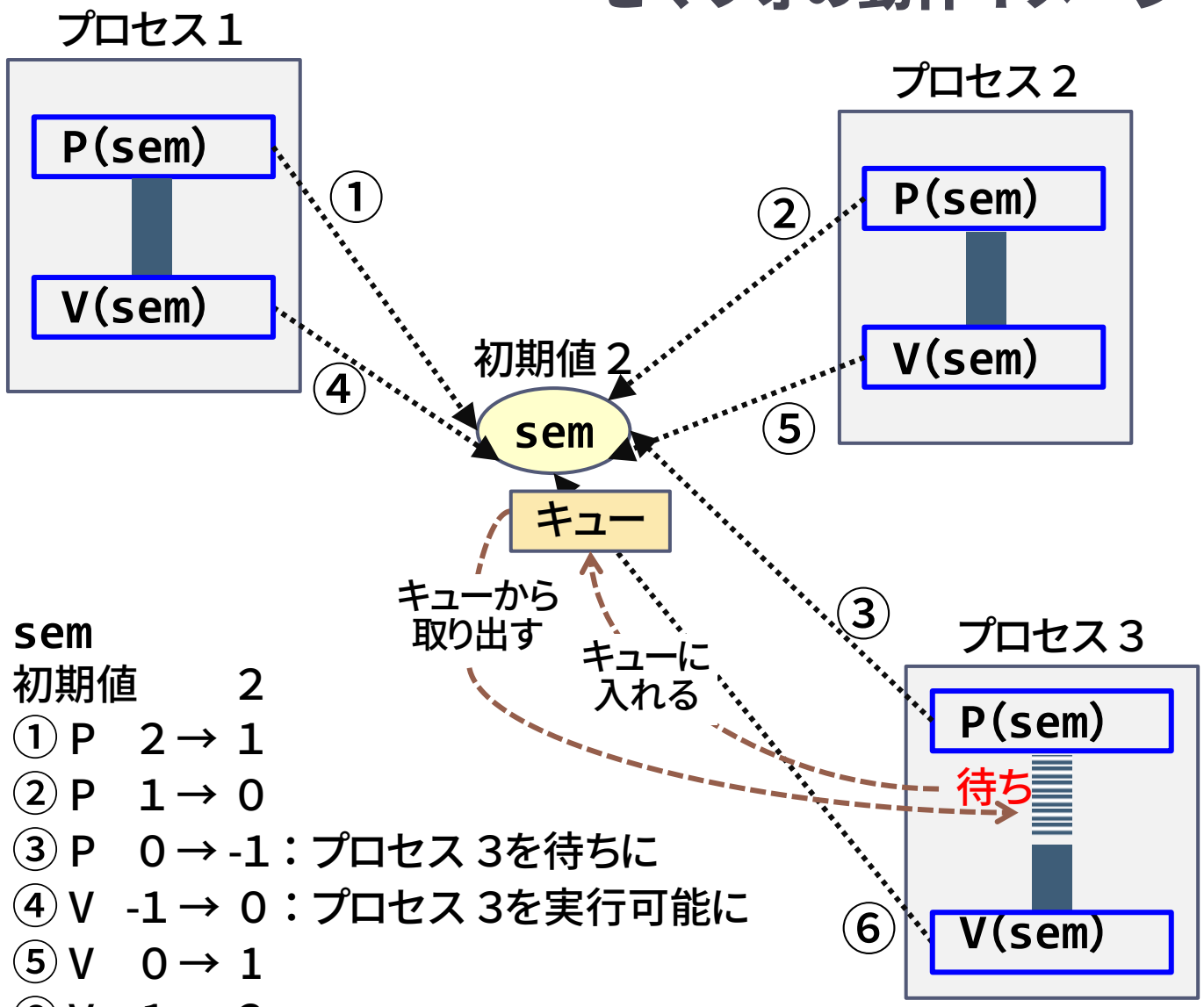
セマフォでは、使用できる複数の資源の数を示すセマフォ変数と、それに対する操作用の関数PとVが用意されます。

P操作は資源を確保しようとするもので、資源が確保できればセマフォ変数を1減らし、確保できない(負になる)ときは、発行したプロセスを待ち状態にします。

V操作は資源を解放するもので、確保されている資源の数を1増やします。この資源を待っているプロセスがあれば、それを実行可能にしてスケジューラに行きます。
待っているプロセスがない場合は、そのままです。前と同じくセマフォ変数に対する操作は不可分でなければなりません。

また、前ページのロック/アンロックはセマフォ変数semの初期値が1の場合と考えられます。(0か1かの2値をとるので、バイナリセマフォと呼ばれることがあります)

# セマフォの動作イメージ




sem
初期値　　2
① P　2 → 1
② P　1 → 0
③ P　0 → -1：プロセス 3を待ちに
④ V　-1 → 0：プロセス 3を実行可能に
⑤ V　0 → 1
⑥ V　1 → 2

3つのプロセスでセマフォによって2つの資源の排他制御を行う例を示します。semの初期値は2です。①②③④の順で実行されるものとします。

プロセス1の①で、P操作によりsemは2から1になります。その後、プロセス2が②でP操作によって確保しようとすると、semは1ですので確保でき0となります。
次にプロセス3が③でP操作すると、semが-1になり確保できず、プロセス3は待ちになります。

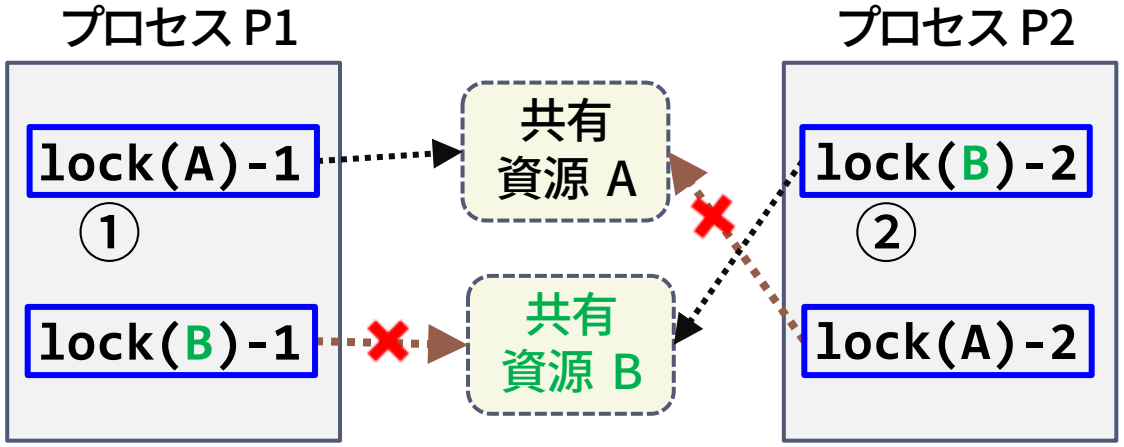
プロセス1が④でV操作により資源を解放すると、semを+1し、キューを見て待っているプロセス3を実行可能にします。
プロセス2が⑤で資源を解放すると、semは1になります。プロセス3が実行され、⑥で解放すると、semは+1されます。

# デッドロック問題

- デッドロック（deadlock）　排他制御において注意すべき点の代表

  複数のプロセスなどの処理単位が互いの処理終了を待ち、結果としてどの処理も先に進めなくなってしまうこと

  （互いに相手の占有している資源の解放を待って、処理が停止してしまう）




- lock(A)-1→①→lock(B)-2→②→lock(B)-1 … P1 は待ち状態に
  →lock(A)-2 … P2 は待ち状態に
- この結果、プロセス P1 と プロセス P2 はお互いが資源を待ちあって永久に待ち状態が解除されなくなる

排他制御における注意点として重要なものに、デッドロックがあります。「膠着状態」に陥っていることです。

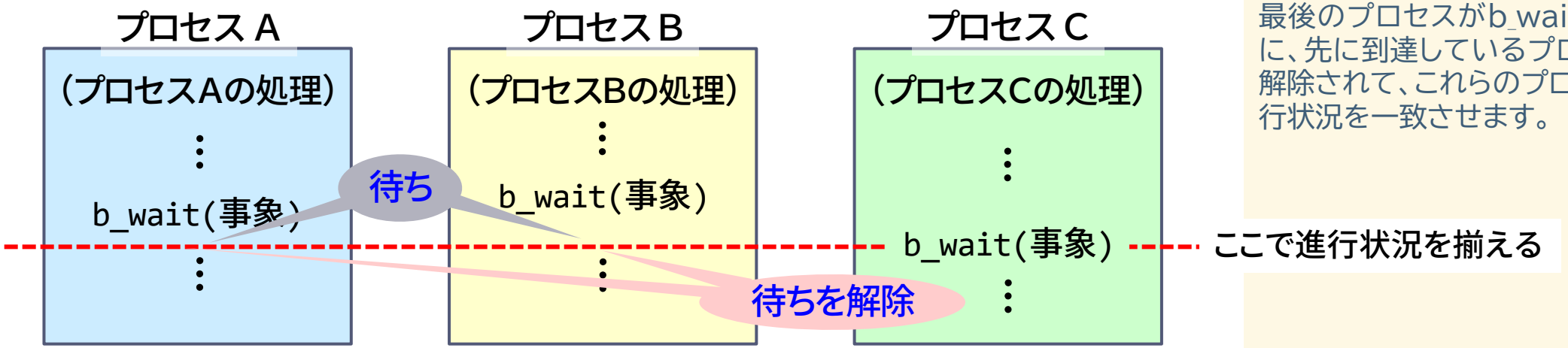
たとえば、2つのプロセスで2つの資源を排他制御したいとき、一方のプロセスP1がある共有資源Aを確保し、他方P2がもう一つの資源Bを確保している状態で、P1がBを確保しようとすると、すでにP2に確保されているので待ちになります。すると、処理が進まないのでP1はAを解放できません。
ここでP2がAを確保に行くと、待ちになります。処理が進まないのでP2はBを解放できません。

ということで両者が資源を確保したまま解放できない状態がずっと続きます。

# プロセスの同期

- **同期**（synchronization）
  複数のプロセスが互いに協調して処理を進める場合、処理手続き上のある地点で他のプロセスなどの処理の進行を一致させること
  協調 … 協力して処理を実行する
  （別のプロセスの処理結果を受け取って先に進むなど）

たとえば、b_wait 関数で同期をとる地点を指定し、2つ以上のプロセスがその地点でプログラムの進行状況を一致させる（先に到達したものが待つ）

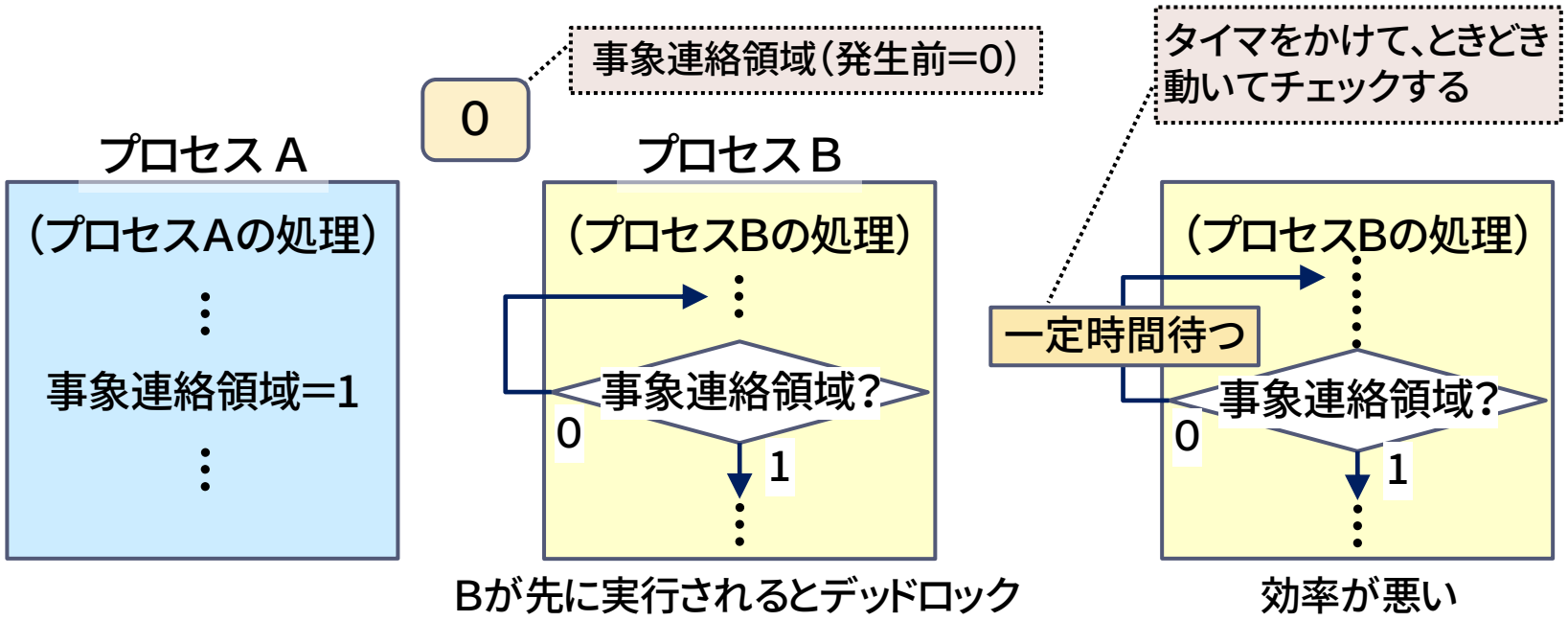



プロセスの同期についてです。
同期は、処理のある地点で他のプロセスの処理の進行をみて、他のプロセスがまだであれば、その完了（ある地点への到達）を待ち合わせることです。

これによって、他のプロセスで行われた計算結果を正しく受け取って次の処理に移ることができます。

先にb_waitに到達したプロセスは、残りのプロセスすべてがb_waitするまで実行を「待ち」とします。
最後のプロセスがb_waitに到達したときに、先に到達しているプロセスの「待ち」が解除されて、これらのプロセスが同期し、進行状況を一致させます。

- 他プロセスでの処理完了を「事象発生」で受け取るまで「事象待ち」する
  - うまくない同期処理の例



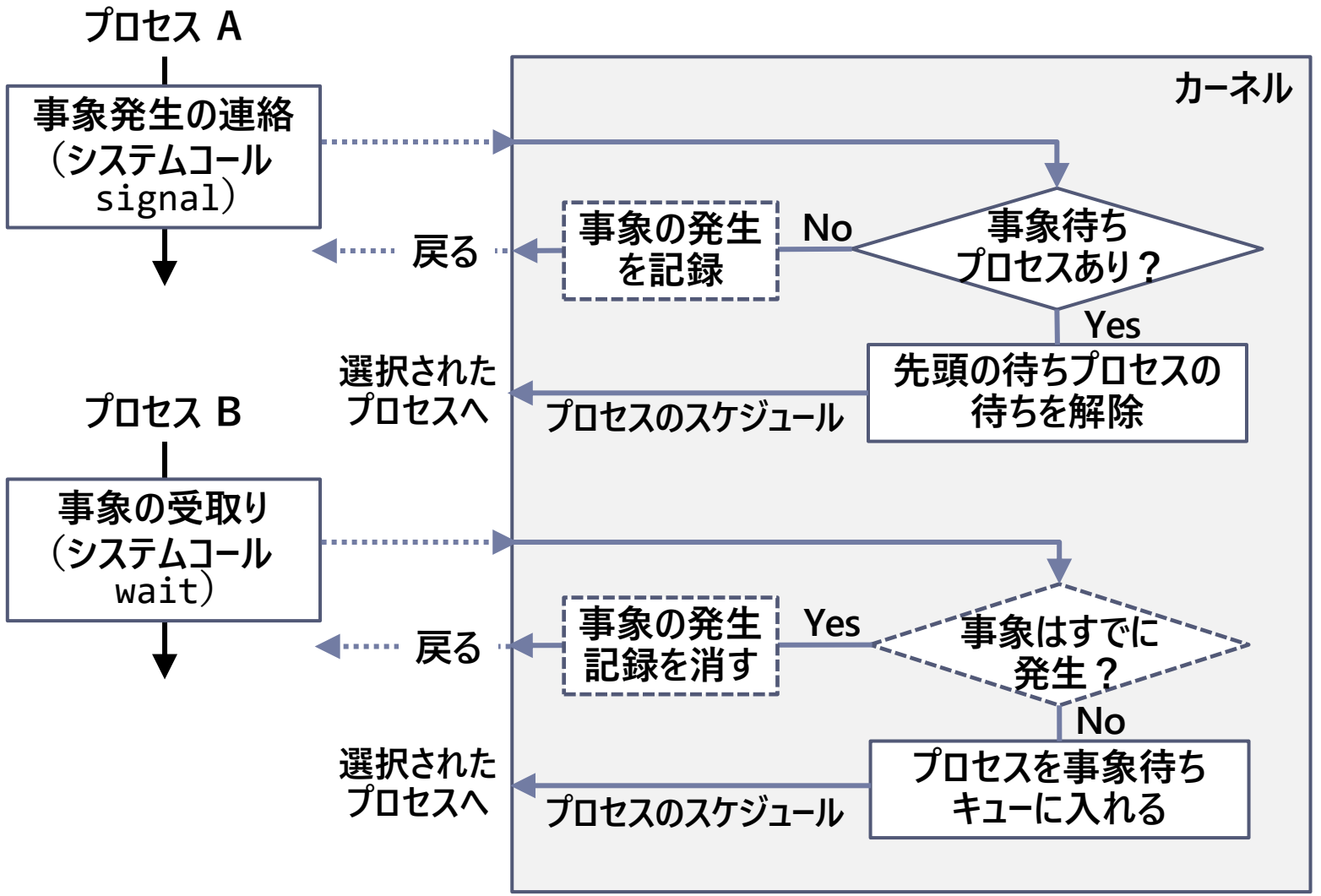

- 同期はシステムコールとなる
  たとえば signal(事象) と wait(事象)

前ページの例はプロセスがある地点で停止し、他の全てのプロセスが到達するまで進行しないようなものですが、ある事象の発生が通知されるまで待つものもあります。

そのような同期処理の実現法として、共有メモリに変数を置いて、必要な処理が終了するとその領域に1を書き、別プロセスではそれが1になるまでビジーウェイトで見ていればよさそうですが、図のように必要な処理が完了する前に見に行くと、デッドロックが起こる可能性があります。

同期の機構はOSとハードウェアによって提供され、同期はシステムコールとなります。
同期を行う方式はいくつかあり、仕様はOSによって異なります。

# 事象連絡のシステムコールによる同期の実現




システムコールによって、事象発生の連絡と受取りを行うようにして同期を実現する仕組みを示します。
事象発生連絡の処理では、左図の上側のように、その事象待ちのプロセスがあれば、先頭のプロセス（キューにつながれているとします）を解除します。なければ、事象が発生していることを記録して戻ります。

事象の受取りでは、事象が発生していればそれを受け取ることができるので、事象が発生していることを消し、戻って処理を続けます。

まだ発生していなければ、そのプロセスを事象待ちのキューにいれて次に移ります。

# セマフォについて補足

- セマフォを用いた同期（事象の連絡）
  - セマフォの初期値を 0 とする
    - V操作が「事象発生の連絡」、P操作が「事象の受け取り」となる

      セマフォが +1 される
- セマフォの問題点
  - P操作、V操作の実行責任はユーザにある
    → 決められた手続きをとらなくてもユーザがクリティカルセクションにアクセスできる
  - P操作と対になるV操作の実行はユーザの責任であり、V操作を行わないことによるプロセス競合やリソース独占の危険性がある

# モニタ

- モニタ（monitor）による排他制御と同期
  モニタは、共有資源とそれに対する操作（確保・解放）、初期化コードが一体となった構造（抽象データ型）
  - モニタへのアクセスはメソッド呼出しを通じてのみ
  - モニタ内で同時に2つ以上のプロセスが実行状態になりえない（相互排除を保証）→ 同期の条件を明示的に記述する必要がない
  - 共有資源が利用可能でないときに、それを要求したプロセスを待ち状態にする機構
    - 条件変数 … 待ち状態の原因を示す
    - signal操作 … 待ち状態にあるプロセスの1つの実行を再開させる
      待ち状態のプロセスが存在しない場合、なにもしない
    - wait操作 … 他のプロセスがsignal操作をするまで、このwait操作を呼び出したプロセスを待ち状態にする
    - signal、wait操作はモニタ内部だけで可能

排他制御の機構にモニタと呼ばれるものがあります。

ビジーウェイトやセマフォでは、共有資源を操作する手続きを明示的に記述する必要があって、エラーの原因（P操作とV操作の順序を間違えるなど）となることもあり、それを解決するためにモニタの概念が考案されました。

同時に2つ以上のプロセスがモニタの中で実行状態にならないようにするので、相互排除が保証されます。

モニタは、共有資源とそれに対する確保や解放などの操作、初期化コードが一体となった構造を持ちます。共有資源に対する同期の命令はそれを操作するモニタ内で指定されるので、エラーがあってもモニタ内に局所化されます。

# モニタによる排他制御と同期

```
monitor class buffer {
 private:
  バッファ; // データの配列
  condition empty, full;

 public:
  データ get(void)
  {
    if (バッファが空) empty.wait();
    バッファ から データ を取り出す;
    full.signal();
  }

  void put(データ)
  {
    if (バッファが満杯) full.wait();
    バッファ に データ を格納する;
    empty.signal();
  }
};
```

局所変数はモニタ内部でのみアクセス可能

モニタにアクセスするメソッド

```
buffer buf;
```

送信側プロセス

```
while(TRUE) {
  送信データの作成;
  buf.put(データ);
};
```

受信側プロセス

```
while(TRUE) {
  受信データ= buf.get();
  受信データの処理;
};
```

モニタによる操作は、その中の仕組みよりもそれを実現している言語からみることでわかります。
C++言語のような構文で記述すると、「monitor」であるクラスがあるものとします。

このクラスのオブジェクトbufを生成すると、それを操作するプロセスが複数あってもモニタの中では1つしか実行できないようになっているので、getやputの実行で自動的に排他制御が実現され、ロックやアンロックをプログラムで記述する必要がありません。

getにおいて、条件変数emptyは、バッファが空のときデータを取り出せないので、emty.wait()でプロセスを待ち状態にします。1つ取り出せば、満杯でなくなるので、full.signal()によって、put時に満杯で(full.wait()によって)待ち状態であったプロセスを再開します。

機構の実現には、不可分命令を用います。

# スレッド

- スレッド（thread）… プログラムの実行単位
  仮想的なCPUをプロセスから分離し、利用するもの
  （「軽量プロセス」とも呼ばれる）

  プロセスにはCPUや仮想アドレス空間などが割り当てられ、その資源内で独立して動く
  - 独立した仮想アドレス空間が不必要な場合、メモリの利用効率が悪い
  - プログラムによっては、共有メモリを利用しながら複数の処理を行った方が楽な場合がある

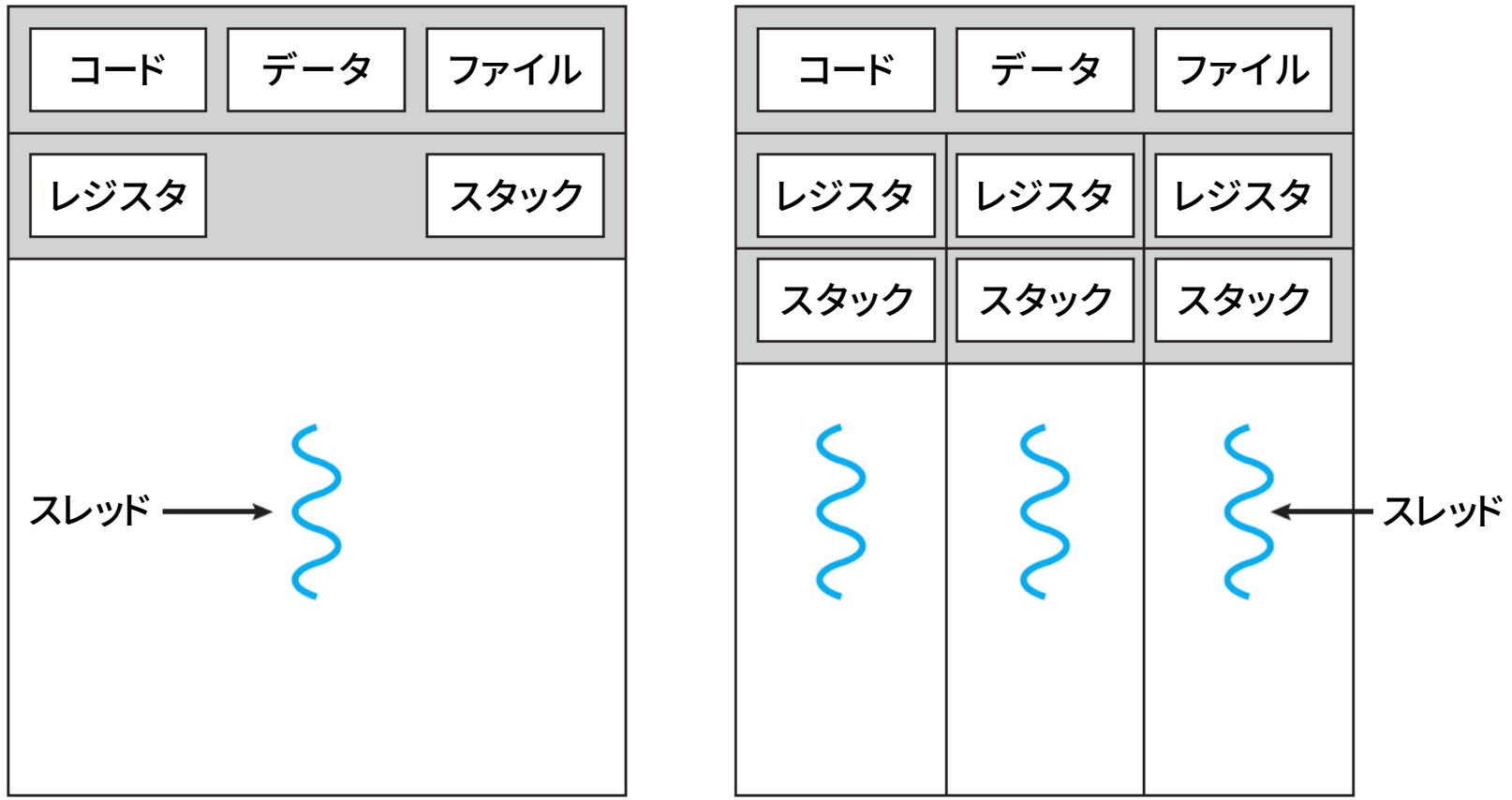
  1つのアドレス空間（プロセス）内に複数の実行単位（スレッド）をもつようにする

- 1つのプロセスは、1つ以上のスレッドから構成される

スレッドとはプロセスと似た概念でプログラムの実行単位になります。
プロセスと何が違うかといいますと、プロセスにはCPUと仮想アドレス空間が割り当てられます。CPUが割り当てられるのはスレッドも同じですが、スレッドには独立した仮想空間は割り当てられません。

独立した仮想アドレス空間があると、保護ができたりするわけですが、反面、共有が難しいという問題もあります。またプロセスの切り替えで仮想アドレス空間を切り替える処理は重いものです。
そこで、独立した仮想アドレス空間が不要な場合、たとえば共有空間を使って多数の関連する計算を並行して行いたいときは、スレッドを使うと効率がよくなります。

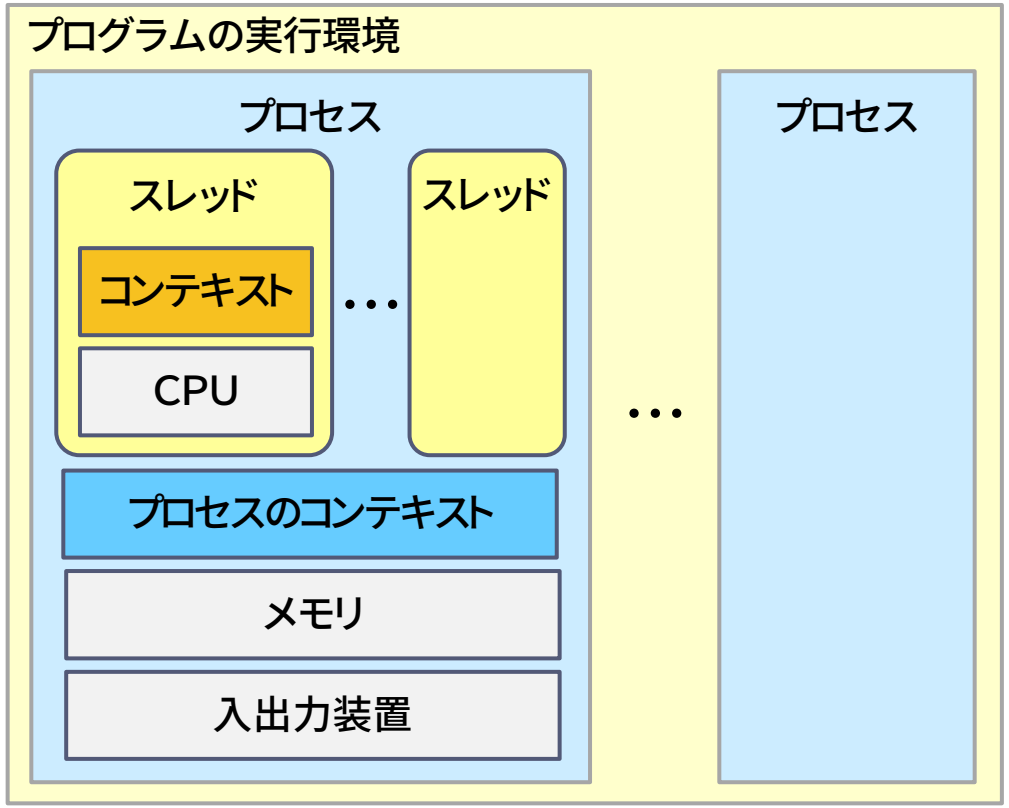
スレッドは「軽量プロセス」と呼ばれることもあります。




参考書「オペレーティングシステムの概念」より

# プロセスとスレッドのイメージ

- 特定のプロセスに属するスレッド群は1つの仮想アドレス空間の全体と仮想化された入出力装置を共有する
  - 同一プロセス内の複数スレッドを同一の仮想アドレス空間上で実行できる




| プロセスごとの項目 | スレッドごとの項目 |
|---|---|
| 1つのアドレス空間 | 1つのプログラムカウンタ |
| 大域変数 | レジスタ群 |
| オープンされたファイル | 1つのスタック |
| 子プロセス | 子スレッド |
| … | 状態 |

複数のプロセスによってプログラムが実行されるわけですが、1つのプロセスの中に、その仮想アドレス空間を共有する複数のスレッドがあります。

スレッドごとにコンテキストがあり、それを用いた実行の切り替えはプロセスのものと同じです。

# プロセス間通信

- プロセス間での情報共有
  - プロセスは独立しており、資源の共有は行われない
    しかし、2つ以上のプロセスが協調しながら動作するためには、
    - 同期　と
    - データの授受

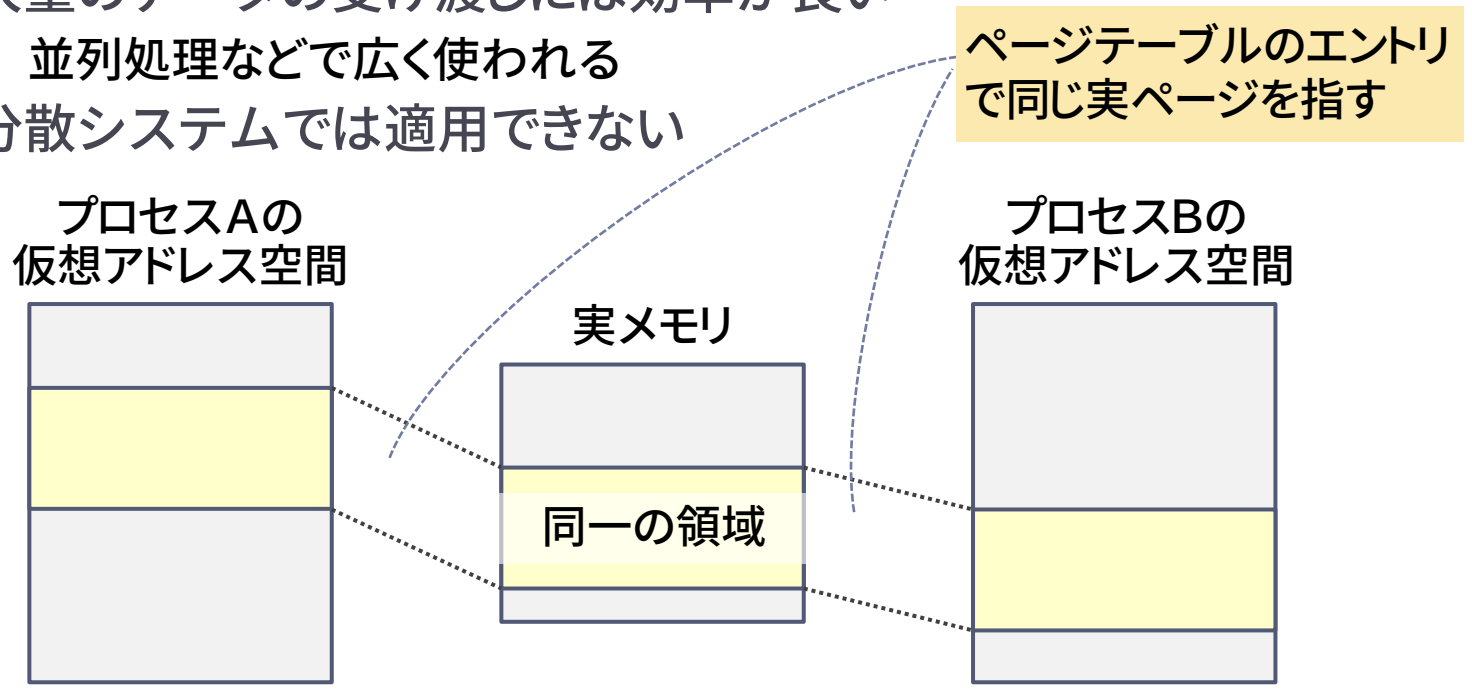
    が必要
  - 複数のプロセス間でデータの授受を行うために、プロセス間通信（Interprocess Communication、IPC）の機構がオペレーティングシステムで提供される
  - このための手段として、次の2つがある
    - 共有メモリ
    - 仮想的な通信路

協調して計算を行うには、プロセス間でデータをやり取りすることが必要です。同期をとって進行状況をそろえることは、p.11～12で見ました。

複数のプロセスの間でデータのやり取りを行うために、プロセス間通信が提供されます。
共有メモリと仮想的な通信路によって実現します。

# 共有メモリによるプロセス間通信

- 共有メモリ
  2つ以上のプロセス間でメモリの一部分を共有してアクセス可能とし、データの交換を行うもの
  - 共有部分の読み書きはプロセスの責任で行う
    - 必要であれば、同期や排他制御を行う
  - 大量のデータの受け渡しには効率が良い
    - 並列処理などで広く使われる
  - 分散システムでは適用できない

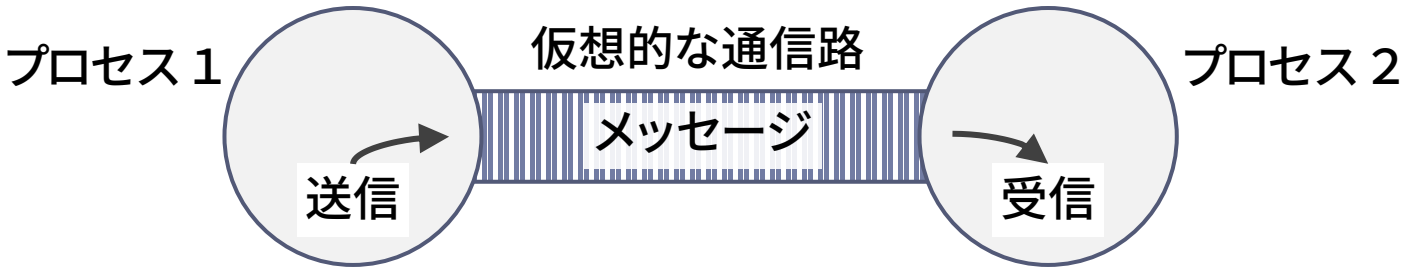



共有メモリは、仮想記憶のところで出てきました。プロセスごとにもつページテーブルのエントリで同じ実ページを指すことで、ページ単位で共有が行えるというものです。

読み書きや同期、排他制御を自身で行うことが必要ですが、大量のデータの受け渡しには効率のよいものです。

# 仮想的な通信路によるプロセス間通信

- プロセス間の仮想的な通信路
  メモリを共有するのではなく、オペレーティングシステムが仮想的な通信路をアプリケーションに提供する
  - 2つのプロセスの間に仮想的な通信路を形成し、それを用いてデータの送受信を行う
  - 多くの場合、データのやり取りだけでなく、同期も行う
  - 通信路により、メッセージの送受信を行う（メッセージパッシング）
  - 通信相手としてプロセスの名前を指定する（直接通信方式）ものと、論理的な通信媒体（チャネルやメールボックスなど）を間接的に指定する（間接通信方式）ものがある

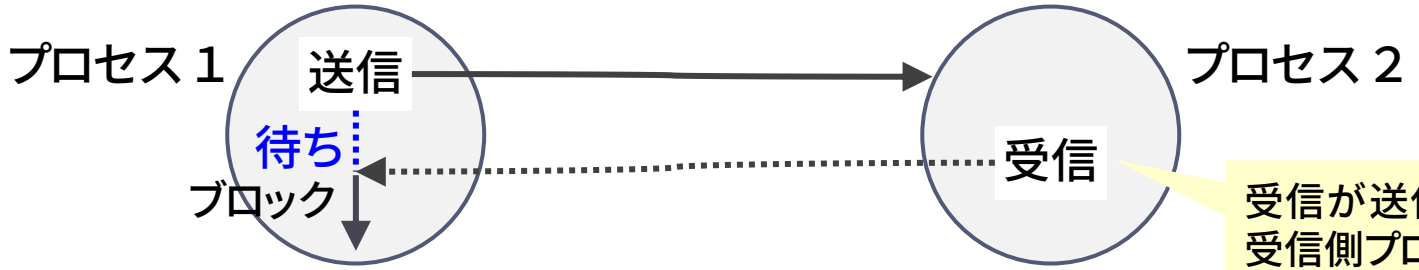



OSで仮想的な通信路を提供する方法もあります。

通信路といってもメモリにバッファとして取られているわけですが、データのやりとりだけでなく同期も行います。
たとえばメッセージでは、それが送信された後でのみ受信可能なので、同期が行われることになります。

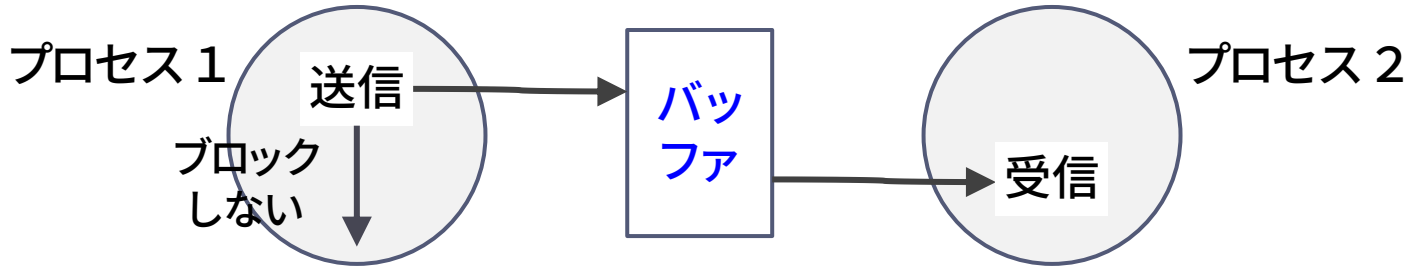
# 同期式通信と非同期式通信

- 同期式通信
  メッセージの受渡しが完了するまで待つ方式（同期型、ブロッキング型）




受信が送信より早く発生した場合は、
受信側プロセスが待つことになる

- 非同期式通信
  メッセージの受渡しが完了する前に呼出し元に復帰する方式（非同期型、ノンブロッキング型）
  - 送受信のデータはバッファに保持される




同期式と非同期式は、ファイルや入出力でも出てきていますが、プロセス間通信でも同じイメージです。

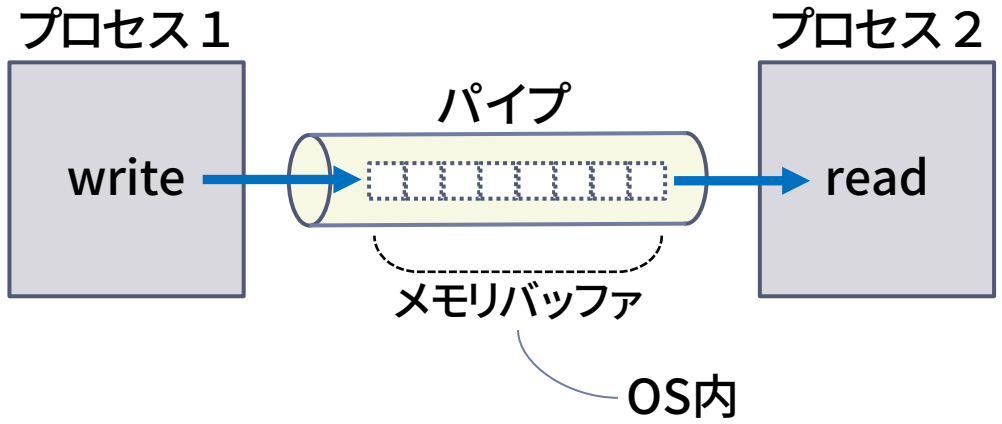
同期式ではデータの受け渡しが確認されるまで呼び出しプロセスがブロックされます。

非同期式では、バッファにいったん入れられて制御がすぐに実行プロセスに返ります。

非同期式で、メッセージが受信されたか否かの確認が必要な場合には、割込みや「完了通知」操作を用いることになります。

# パイプ

- UNIX系のオペレーティングシステムやWindowsでは同一の計算機上のプロセス間の通信路をパイプ（pipe）と呼ぶ
  - パイプでは、データがバイト単位で書き込まれた順に読み出される
  - パイプは通常の入出力とプロセス間通信のインターフェースを統一したところに特徴がある
    （通常の読み出し/書込み操作を使って通信が行える）
  - パイプは一種のファイルであり、実体はメモリで、それを二つのプロセスが共有する




UNIXなどでは同一のコンピュータのプロセス間での通信路をpipe（パイプ）と呼びます。
ファイルの入出力がバイト単位で、プロセスの出力を他のプロセスの入力につないだというイメージです。

実体はメモリに確保されたバッファで、出力側がバッファに書いたものを入力側が順に読み出して処理することになります。入力側と出力側で処理速度が違ってバッファがいっぱいになったり空になったりしても送信側/受信側が待つことで同期がとられます。

# 教科書との対応

- TAS命令については、教科書の Column マルチプロセッサのカーネルに記述のある「メモリの内容の判定と変更を1命令でできる命令語」です
- セマフォのP操作、V操作について、教科書ではそれぞれ DOWN、UP となっています
  教科書 p.110 注*3 に「P と S という名前」とありますが、「P と V という名前」の誤植と思われます

モニタは、非常に小さなレンタルDVD店にたとえることができます。

店には通常一人の客しか入ることができず、中に客がいる場合、他の客は行列を作って待ちます。店に入れた客は、中で借りたいDVDを探し、あれば借りて店を出ます。客が店を出ると待っていた客が(もしあれば)店に入ります。
DVDが店にない場合、借りたいDVDの棚に自分のネームプレートを置いて、店に備え付けのベッドで眠って待ちます(wait)。すると、次の客が店に入れるようになります。

DVDを返す場合にも店に入ります。DVDを返す際に棚を見て、もしネームプレートが置いてあれば、その客を(眠っている客は複数かもしれない)起こします(signal)。店は小さいので起こされた客はすぐには動き出すことはできず、起こしてくれた客が出てから動き出してDVDを借りることになります。

店はモニタに、DVDは共有資源に相当します。店内に一人しか入れないという制約が、モニタには高々一つのプロセスしか入れないことに対応します。

# 第9回の課題

- デッドロックが起きないようにするにはどのような点に気をつければよいか、p.10の例にもとづいて説明せよ

今回の課題です。クラスウェブのレポートで提出してください。

期限は、6/15の午前中とします。

# 事後学習・事前学習

- 今回の講義資料に基づいて内容を振り返り、教科書などの該当箇所を読む
- 教科書第12章（12.1、12.2）に目を通す

今回の講義内容の振り返りと次回の準備をお願いします。