2024 機械情報夏学期
ソフトウェア第二

担当：岡田 慧 k-okada@jsk.t.u-tokyo.ac.jp

10. オペレーティングシステム（プロセスとスレッド）

1　プロセッサ資源の有効利用

配線盤からバッチシステムへ　真空管を使った初期のコンピュータは，プログラムは配線を組みなおすことで行われていた．OS という言葉も無かった．配線を組み替える間 (プログラムを入力する間) は，演算回路は使用できなかった．1950 年代半ばにトランジスタが導入され，パンチカードに打たれたプログラムはジョブという単位で管理され，複数のジョブをテープにまとめて記録し，バッチというシステムがそれらを順に実行しそれぞれのジョブの結果を出力テープに書き出すというものになった．このバッチシステムが OS の原型であり，プロセッサという計算資源を遊ばせておく時間をなるべく少なくし有効利用するためのシステムであった．つまりプログラム入力の間にも，プロセッサを働かせておくことができるようになったのである．

マルチプログラミング　バッチシステムでは，プロセッサが入出力操作の完了を待っている時間などに，プロセッサがアイドルになっている時間が多くあった．これを解決するために考え出されたアイディアがマルチプログラミングであった．マルチプログラミングは，メモリをいくつかのパーティションに分けて，それぞれのパーティションに別々のジョブ (プログラム) を入れておき，あるジョブが入出力完了を待っている間，他のジョブが CPU を使用できるようになった (図 1)．他のジョブ (プロセス) が待ちに入らなくても，ある時間が経つとプロセスを切り替えるシステムが時分割方式である．メモリ上に複数のプログラムが置かれるようになり，メモリの管理も OS の役割の重要な要素になった．

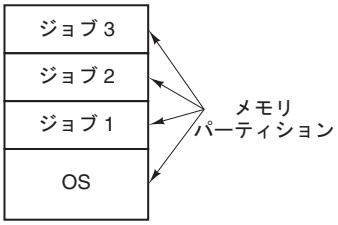

図 1: メモリ内に 3 つのジョブがあるマルチプログラミング

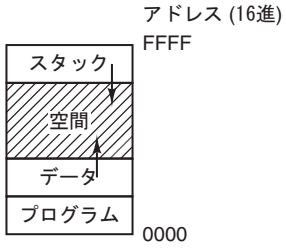

図 2: プロセスは，プログラム，データ (ヒープとも言う)，スタックの 3 つのセグメントを持っている．

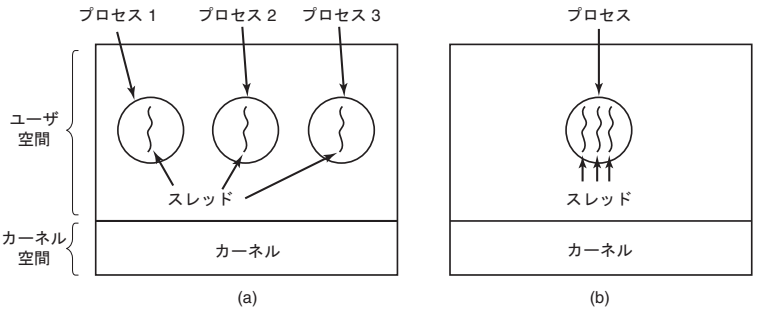

図 3: (a)1 つのスレッドを持つプロセスが 3 つ，(2)3 つのスレッドを持つプロセスが 1 つ

1.1 プロセス

プロセス (process) は，基本的には，実行中のプログラムである．それぞれのプロセスにはアドレス空間 (そのプロセスがアクセスできるメモリ空間) があり，そこにはプログラム，データ，スタックがある (図 2)．また，各プロセスは，固有のプロセス識別子 (process ID: **PID**)，プログラムカウンタ，スタックポインタ，レジスタセット等，プログラム実行に必要な全ての情報を保持している．プロセスの生成は親プロセスが子プロセスを生成することで行われ，プロセス木を構成する．

1.2 スレッド

1 つのプロセスの中に複数の処理の流れを持つことができる．この処理の流れの 1 つ 1 つをスレッド (thread) と呼ぶ (図 3)．同一のプロセスの中の複数のスレッドは，同一のメモリ空間を利用する．スタックは各スレッドで独立している．スレッドはプロセスの特徴のいくつかを持っているので，軽量プロセス (lightweight process; **LWP**) と呼ばれることもある．複数のスレッドを使うことを，マルチスレッド (multithread) と呼ぶ．

1.3 スケジューリング

プロセッサの数より多いプロセスやスレッドが実行中の場合にどのプロセス・スレッドを実行するかを決めるのが，スケジューラ (scheduler) である．スケジューラによって，例えば，図 4 のように複数のプロセス・スレッドが短時間ずつプロセッサ資源を使用する．

プロセス・スレッドには，実行中 (running)・実行可能 (ready)・待ち (blocked)[1]の三種類の状態がある．実行可能状態のプロセス・スレッドはキュー (queue) になってスケジューラが CPU を割り当ててくれるのを待っている．CPU が実行するプロセス・スレッドの切り替えを，コンテクストスイッチ (context switch) と言う．

また，実行中のタスクを一時的に中断する動作をプリエンプションと呼ぶ．しかしながら、カーネルの機能や割り込み処理は等にはプリエンプションが不可能な操作が存在する．これらが終了するまでプリエンプション出来ないようにしておかないと，競合状態からデッドロックが発生する．

[1]待ち (ブロック) の状態は，入出力の完了を待つような場合に生じる．例えば `read` システムコールを実行し入力が来る前だと，待ちの状態になる．

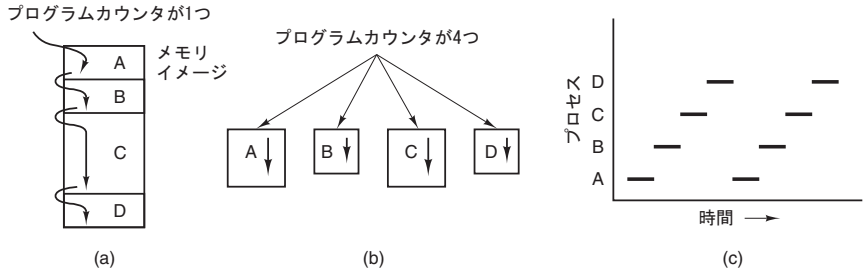

図 4: (a)4 つのプログラムによるマルチプログラミングのメモリイメージ，(b)4 つの独立な逐次プログラムの概念的モデル，(c)1 つのプロセッサで 4 つのプロセスを実行する場合の実行例

一方でカーネルモードのタスクであってもプリエンプション可能なシステムはプリエンプティブ・カーネル等と呼ばれる．

スケジューラのスケジューリング方式には様々な種類がある．代表的なスケジューリング方式には，到着順 (first-com first-served)，最短ジョブ優先 (shortest job first)，最小残り時間優先 (shortest remaining time first)，ラウンドロビン[2](round-robin scheduling)，優先度スケジューリング (priority scheduling) 等がある．

Linux のスケジューリングは循環キューを用いたラウンドロビンが基本である．カーネルバージョン 2.4 では実行可能状態のキューから，優先度が最も高いプロセスを CPU に割り当てるようになった．

カーネルバージョンが 2.6-2.6.22 (2003-2006) までは，優先度付きキューを利用し，常にこのキューの先頭を取り出す O(1) scheduler と呼ばれるものが利用されていた (`https://en.wikipedia.org/wiki/O(1)_schedule`)．これは 静的な優先順位に加えて，プロセスの平均スリープ時間を考慮した動的な静的順位を考慮してクオンタム（割り当てる時間）を算出する．これにより，ユーザからの入出力によるスリープ時間を多く有するプロセス (interactive process) はより多くの時間を割り振られる．

カーネルバージョン 2.6.23 以降（2007-) は CFS(Completely Fair Scheduler) が採用されている．プロセスはそれぞれ，それまでに消費した実行（プロセッサ）時間をキーとした赤黒木（平衡二分木の一つ）で管理され，最も消費した実行（プロセッサ）時間が短いプロセスを選択している．また，スリープした時間が長い場合は実行時間が少ないため自動的に優先度が上がるようになっている (`https://en.wikipedia.org/wiki/Completely_Fair_Scheduler`)．

1.4 リアルタイムスケジューリング

リアルタイムシステム (real-time system) は，時間が本質的な役割を担うシステムである．リアルタイムシステムでは，結果の正しさに加えて，結果を出すまでの時間が，結果の価値を決める．ここでの時間とは，早いか遅いかではなく，デッドライン以内か否かである．ハードリアルタイムとはデッドラインを過ぎてしまうと価値が無くなるようなシステム (例:エアバッグ，天気予報) で，ソフトリアルタイムとはデッドラインを過ぎると価値が下がっていくようなシステム (例:動画再生，蕎麦屋の出前) である．

[2]ラウンドロビンスケジューリングでは，各プロセスはクオンタム (quantum) と呼ばれるある時間を割り当てられ，CPU を割り当てられたプロセスはその時間を使い切るか待ち (blocked) の状態になるかすると CPU の切り替えが発生する．

他のプロセス・スレッドが実行中に，別のプロセス・スレッドが CPU を横取り (preemption) できるようなスケジューリング方式を preemptive と呼ぶ．多くのリアルタイムシステムのスケジューラは，プリエンプティブである．

代表的なリアルタイムスケジューリングには，レートモノトニックスケジューリング (rate monotonic scheduling; RMS) と，earliest deadline first (EDF) である (各自それぞれの特徴を調べてみよう)．図 5 及び図 6 は RMS と EDF のスケジューリングの比較を示す．

リアルタイム OS は，リアルタイムスケジューリングの機能を持ち，優先度逆転の問題を防ぐための優先度継承等の機構を備えている．ここでは詳しくは扱わないが，余裕があれば色々と調べてみると良い．

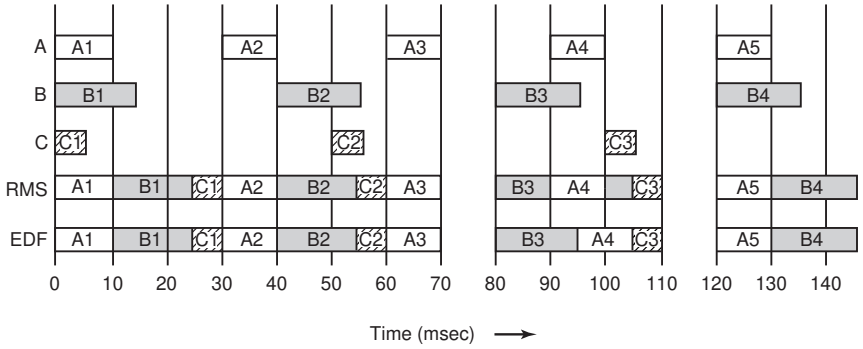

図 5: リアルタイムスケジューリングの例．A,B,C の各プロセスは，30,40,50[msec] 毎に実行可能になり，次にそのプロセスが実行可能になる時刻がデッドラインである．RMS の方が EDF よりコンテクストスイッチが多い．

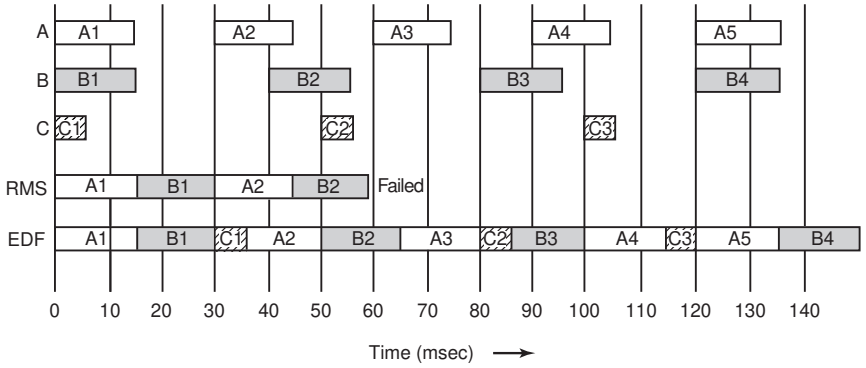

図 6: RMS は条件を満たせば最適なスケジューリングを行うが，CPU 利用率が高い時は失敗する場合が多い．EDF は最適ではないが CPU 利用率が 100%以内なら常にうまくいく．

2　メモリ管理

メモリにはアクセス速度と容量のトレードオフ (trade-off) が存在する (図 7)．理想の高速で大容量のメモリに見せるために，キャッシュ(cache) や仮想記憶 (virtual memory) の仕組みがある．キャッシュの制御はプロセッサ内部で行われ，インストラクションセット (機械語) 以上のレイヤからは隠蔽[3]されている．これに対し仮想記憶は，アクセスされていないアドレス領域の内容をディスクに書き込んでおくことで，主記憶の容量が実際より大きいかのように使える仕組みだが，これは OS の機能として提供される．

単純なスワッピング (swapping) は，プロセス全体をメモリからディスクへ移したり (スワップアウト)，ディスクからメモリに戻したり (スワップイン) する．仮想メモリは，1 つのプロセスの一部分を実行部分付近をメモリに置き，非実行部分をディスクに置く．

仮想メモリを使用するコンピュータでは，メモリアクセスは仮想アドレス空間のアドレスにより扱われる．仮想アドレス空間はページと呼ばれる単位に分割されており，メモリとディスクの間の転送はページ単位で行われる．**MMU**(memory management unit) は，仮想アドレスを物理アドレスに変換する．実メモリ上に無いアドレスにプロセスがアクセスすると，CPU はページフォールト (page fault) というトラップ[4]を発生し，OS はそれを受けて最もアクセスの無かったページをディスクへ書き出し，アクセスのあった仮想アドレスに対応する物理アドレスを含むブロックをディスクから実メモリにロードする．そして MMU に更新を伝え，トラップされた命令から再実行する．

仮想アドレスと物理アドレスの関係は，ページテーブル (page table) と呼ばれる表によって記憶される．ページテーブルは，ページサイズ (例えば 4KB) と仮想メモリ空間サイズ (例えば 4GB(32bit)) の大きさの比率 (この例えなら約 100 万) が非常に大きいので，テーブル自体のサイズも非常に大きくなる．ページテーブルはメモリに置かれるが，メモリアクセスのたびにアドレス変換のためにページテーブルへのアクセスが必要になるので，一度のメモリアクセスプログラムの実行に，時間のかかる主記憶へのアクセスが何度も発生してしまう．そこで，**TLB**(translation lookaside buffer) と呼ばれる高速・小容量のデバイスを MMU 内に設け，ページテーブルの部分エントリをいくつか記憶しておく．TLB はキャッシュと同様に，TLB に無いエントリが必要になると，最もアクセスの無いエントリを破棄し，主記憶からページテーブルの必要なエントリをコピーする．

更に，近年は 64bit のシステムも増え，逆引きページテーブル (inverted page table) を使用して使用メモリ量に応じたテーブルサイズになるような工夫もされている．

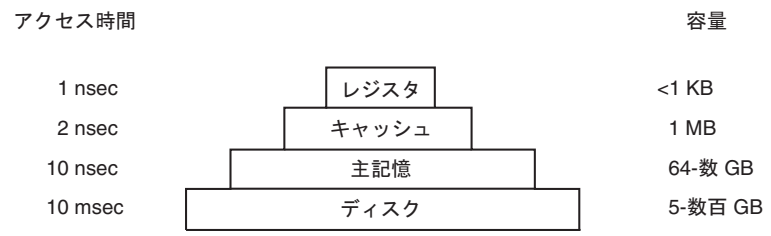

図 7: メモリ階層 (memory hierarchy):アクセス時間と容量はトレードオフの関係

[3] このように階層構造の下の段 (レイヤ, 層) の機能により実現され上の段からは詳しくは見えない (見る必要が無い) ようにすることを，抽象化 (abstraction) という．

[4] 意味は各自調べよう．

3 その他のリソース管理

3.1 ファイルシステム

OS は，ディスクや入出力デバイス装置の特質を隠蔽し，装置に依存しないファイルの生成，削除，読み出し，書き込みのシステムコールを提供する．ファイルの読み出し・書き込みの前後に，オープン，クローズのシステムコールも必要である．また，ファイルをグループ化する方法としてディレクトリの概念を持っている．ファイルのオープンに成功するとシステムコールはファイル記述子 (file descriptor) と呼ぶ整数を返す．その後の操作は，ファイル記述子を用いて行う．

UNIX のファイルシステムで重要な概念は，マウントである．あるデバイスをマウントすると，元からあるディレクトリツリーの指定された枝に，そのデバイスの持つディレクトリ構造を接続することができる．

3.2 入出力・デバイスドライバ

プリンタ・マウス・音声・通信などの入出力デバイスは，UNIX ではデバイスファイルを介してアクセスする．デバイスファイルは，`/dev/`の下にあり，アクセス方法の違いでブロック型デバイスとキャラクタ型デバイスがある．デバイスファイルをオープンすると，そのデバイスを扱うためのデバイスドライバというソフトウェアを介して，デバイスの使用開始の準備を整える．そしてプログラムがそのファイルへの読み書きを行うことで，デバイスドライバはそのデバイスの操作を行う．

3.3 UID

OS はユーザアカウントの機能を提供する．ユーザアカウントには固有の **UID**(user ID; ユーザ識別子) が割り当てられる．プロセスの所有者，ファイルの所有者は，UID で表され，他の UID のアカウントからは許可されたアクセスしかできない．各ユーザはグループのメンバーになれ，グループごとにグループ識別子 (**GID**) が割り当てられる．ファイルやディレクトリへのアクセスの許可・不許可 (パーミッション; permission) は，Read/Write/eXecute を意味する rwx で指定される．例えば，シェルのコマンドである `ls -l` はパーミッション情報を表示し，`chmod` はパーミッションを変更する．

3.4 シェル

シェルは，OS の一部ではないが，OS の機能を頻繁に使用する．sh，csh，tcsh，bash，ksh などの多くのシェルがある．一番元のシェルは sh である．

4　プロセスとスレッド

プロセスは，基本的には，実行中のプログラムである．それぞれのプロセスにはアドレス空間 (そのプロセスがアクセスできるメモリ空間) があり，そこにはプログラム，データ，スタックがある．また，各プロセスは，固有のプロセス識別子 (process ID: **PID**)，プログラムカウンタ，スタックポインタ，レジスタセット等，プログラム実行に必要な全ての情報を保持している．プロセスの生成は親プロセスが子プロセスを生成することで行われ，プロセス木を構成する．

一方スレッドは，1 つのプロセスの中に複数の処理の流れを持ったものである．同一のプロセスの中の複数のスレッドは，同一のメモリ空間を利用する．スタックは各スレッドで独立している．スレッドはプロセスの特徴のいくつかを持っているので，軽量プロセス (lightweight process; **LWP**) と呼ばれることもある．複数のスレッドを使うことを，マルチスレッド (multithread) と呼ぶ．

4.1　プロセスの生成

プロセスの生成には fork というシステムコールを使う．システムコールとは、オペレーティングシステム (OS)（より明確に言えば OS のカーネル）の機能を呼び出すために使用される機構．システムコールは特殊な命令を使うことが多く、それによって CPU は高い特権レベルのコードに制御を渡す．POSIX および類似のシステムでの主要なシステムコールとしては、open、read、write、close、wait、execve、fork、kill などがある。最近のオペレーティングシステムは数百のシステムコールを持つ。例えば、Linux は約 300 種類のシステムコールを持つ (`https://linuxjm.osdn.jp/html/LDP_man-pages/man2/syscalls.2.html`).

生成した子プロセスの終了は wait を使って待つことができる．

これらの関数の説明は man コマンドを用いてマニュアルを表示するとよい．. man -k X とすると，X というキーワードを含む項目を検索し，man -s Y X とするとセクション Y の X という項目を表示する．man -s 1 wait, man -s 3 wait がある．あるいは，インターネットで `man wait` と検索してもよい．

```
pid_t fork(void )
     成功した場合、親プロセスには子プロセスのプロセス ID を返し、子プロセスには 0 を返します。
     失敗した場合には親プロセスに-1 を返し、子プロセスは生成されません。
pid_t wait(int *status)
     子プロセスのいずれかが終了するまで呼び出し元のプロセスの実行を一時停止する。
     以下の二つの呼び出しは等価である:
```

```
/* test-fork.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* exit */
#include <sys/wait.h> /* wait */

int main () {

  int shared_resource = 0;

  pid_t pid;
  pid = fork();
  if ( pid == 0 ) {    /* 子プロセス */
    int i;
    for (i = 0; i < 10; i++ ) {
      printf("Process [%d]: %d\n", pid, i);
      shared_resource++;
      sleep(1);
    }
    printf("Process %d finished, return %d.\n", pid, shared_resource);
    exit(0);
  } else if ( pid > 0 ) {    /* 親プロセス */
    int i;
    for (i = 0; i < 20; i++ ) {
      printf("Process [%d]: %d\n", pid, i);
      shared_resource++;
      usleep(300*1000); // 300 msec
    }
    int status;
    wait(&status);
    printf("Process %d finished, return %d.\n", pid, shared_resource);
  }
  return 0;
}
```

このプログラム test-fork.c をコンパイルをするために，以下のような Makefile を同じディレクトリに作るとよい．これで，make というコマンドを入力すれば，test-fork.c というファイルが test-fork というファイルより新しければコンパイルを実行してくれる．

Makefile の一般的な注意点として 2 行目の行頭は Space ではなく，Tab キーを打つこと．

```
test-fork: test-fork.c
        gcc -o test-fork test-fork.c
```

4.2 スレッドの生成

スレッドには様々な実装が存在するが，現在標準になっているのは POSIX 標準[5] の Pthread と呼ばれるものである[6]．Pthread によるスレッド生成では pthread_create，pthread_join，thread_exit という関数を主に使う．

[5] Portable Operating System Interface：IEEE が策定した異なる UnixOS で移植性の高いアプリケーションを実装するための共通 API

[6] Windows では Win32 API で定義されている Windows スレッドが標準である

```
int pthread_create(pthread_t * thread, pthread_attr_t * attr,
                   void *(*start_routine)(void *), void * arg)
      pthread_create は呼び出しスレッドと並行して実行する、新しい制御スレッ
     ドを生成する。新しいスレッドは、 arg を第 1 引数とする start_routine
     という関数になる。新しいスレッドは、pthread_exit(3) を呼び出すことに
     よって明示的に終了するか、関数 start_routine から返ることで暗黙的に終
     了する。

pthread_t * thread     pthread_create により新に生成されたスレッドを識別するためのポインタ
pthread_attr_t * attr, スレッド属性オブジェクトという構造体へのポインタ．特に属性を指定する
                       必要がないときは NULL により，デフォルト属性を指定することができる．
void *(*start_routine)(void *)  スレッドが実行する関数へのポインタ
void * arg                 スレッドが実行する関数へ渡される引数へのポインタ
```

```
int pthread_join(pthread_t th, void **thread_return)
   pthread_join は、呼び出しスレッドの実行を停止し、 th で指定したスレッド
   が pthread_exit(3) を呼び出して終了するか、取り消しされて終了するのを待つ。

pthread_t th
   th で指定したスレッドが終了するまで実行をサスペンドする．

void **thread_return
   th の帰り値を status が参照する変数に格納する．NULL を指定した場合返り値
   はどこにも格納されない
```

```
void pthread_exit(void *retval);
    呼び出しスレッドの実行を終了する

void *retval
    スレッド終了に伴う返り値をセットする．この値は pthread_joint で受け取
    ることができる．
```

スレッドのサンプルプログラム test-thread.c を以下に示す．

```
/* test-thread.c */
#include <stdio.h>
#include <stdlib.h> /* exit */
#include <unistd.h> /* usleep */
#include <pthread.h>

long int shared_resource = 0;

void *task (void *arg) {
  long int i, loop = (long)arg;          /* 引数の取得 */
  for(i=0;i<loop;i++){
    printf("Thread [%x]: %ld\n", (int)pthread_self(), i);
    shared_resource++;
    usleep(10*1000*1000/loop);
  }
  return (void *)(shared_resource);
}
```

```
int main () {
  pthread_t thread1, thread2;
  long int loop1 = 10, loop2 = 20;

  pthread_create(&thread1, NULL, task, (void *)loop1); /* thread1 を生成 */
  pthread_create(&thread2, NULL, task, (void *)loop2); /* thread2 を生成 */

  int ret1, ret2;
  pthread_join(thread1, (void **)&ret1);
  printf("Thread [%x]: finished, return %d.\n", (int)thread1, ret1);
  pthread_join(thread2, (void **)&ret2);
  printf("Thread [%x]: finished, return %d.\n", (int)thread2, ret2);

  return 0;
}
```

pthread を使ったプログラムではソースコードで`#include <pthread.h>`とヘッダファイルをインクルードし，コンパイル時に `gcc -o test-thread test-thread.c -lpthread` としてライブラリをリンクする．Makefile は以下のように書く．

```
test-thread: test-thread.c
        gcc -o test-thread test-thread.c -lpthread
```

4.3 排他制御

スレッド間ではメモリ空間を共有するため，グローバル変数もまた共有することができる．これは，並行プログラミング[7] を容易にするが複数のスレッド間で同時に同じ変数にアクセスしプログラマの意図とは異なる振る舞いを起こすことがある．

このように，単一のリソースに対して，複数の処理が同時期に実行されると破綻をきたすコード上の領域をクリティカルセクション (Critical section) とよび，システムの他の部分から見て，この領域をひとつの操作に見えるアトミック操作（Atomic Operation）になるような排他制御が必要になる．ミューテックス (Mutex) とは、このようにクリティカルセクションでアトミック性を確保するための同期機構の一種である．

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
*mutexattr)
     mutex が指す mutex オブジェクトを、 mutexattr で指定された mutex 属性オブ
     ジェクトに従って初期化する。 mutexattr が NULL, ならば、デフォルトの属性
     がこのかわりに使われる。

int pthread_mutex_lock(pthread_mutex_t *mutex)
     与えられた mutex をロックする。mutex が現在ロックされていなければ、それは
     ロックされ、呼び出しスレッドによって所有される。この場合
     pthread_mutex_lock は直ちに返る。mutex が他のスレッドによって既にロックさ
     れていたのならば、pthread_mutex_lock は mutex がアンロックされるまで呼び
     出しスレッドの実行を停止させる。
```

[7] 並列処理 (parallel processing): 「物理的に」多数のプロセッサによって実行される．並行処理 (concurrent processing): 「論理的に」複数のタスクが実行される．実際にプロセッサが一個でも複数でも構わない．従って, 並行処理を並列処理で実現することもできる．なお, 並行処理という用語は, 主に, OS, データベースの分野で用いられる．

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
     与えられた mutex をアンロックする。 pthread_mutex_unlock の開始時点で、こ
     の mutex は呼び出しスレッドによりロックされ所有されているものと仮定される。
```

例えば先程の test-thread.c プログラムで shared_resource++ の代わりに以下のようなプログラムに変更した test-thread2.c プログラムとするとクリティカルセクションの排他制御が必要になる.

```
    //shared_resource++;
    //usleep(10*1000*1000/loop);
    int j, tmp;
    for(j=0;j<10000;j++){
      tmp = shared_resource;
      tmp = tmp + 1;
      usleep(1);
      shared_resource = tmp;
    }
    usleep(10*1000*1000/loop-10000);
```

4.4 シグナルによるプロセスの終了

端末で C-c と入力することで SIGINT シグナルが発生し，プログラムは signal システムコールに対して登録された関数これを受け取り終了操作を行う．このようにシグナルを受信してそのシグナルに対応した処理をする関数をシグナルハンドラと呼ぶ．シグナルはゼロ除算や不正メモリ参照でも生じる.

```
void (*signal(int sig, void (*func)(int)))(int)
    signal()) システム・コールは sig で指定された番号のシグナルに新しいシグナ
    ル・ハンドラーを設定する。このシグナル・ハンドラーは func に設定される。
    func はユーザの指定した関数，SIG_IGN，SIG_DFL のいずれかである.
```

```
#include <signal.h>

static void signal_handler(int sig) {
  printf("signal handler for %d\n", sig);
  exit(1);
}

int main () {
  signal(SIGINT, signal_handler);
  ....
}
```

シグナルは kill コマンドで発生させることができる.

```
kill -INT [プロセス番号]
```

とするのは，端末で C-c を押したことと同じことである.

シグナル一覧は man -s 7 signal で見ることができる．代表的なものを以下に示す.

```
 1   SIGHUP    デーモンプロセスに設定の再読み込みをさせるのに良く利用される
```

2 SIGINT キーボードからの割り込み (Ctrl-C)
9 SIGKILL Kill シグナル。kill -kill [PID]
14 SIGALRM アラーム。一定時間後に SIGALRM を飛ばせる
15 SIGTERM 終了シグナル。kill -term [PID]

4.5 Python におけるマルチスレッド

Python では threading モジュールによりマルチスレッドや排他制御が可能になる．例えば，前述の C 言語のプログラム例は以下のように書くことができる．

ここでは threading.Thread クラスのコンストラクタにおいて，target で呼び出す関数（オブジェクト），args で target を呼び出す引数を与えることができる．スレッドは start() で開始し，join() で終了までを待機することができる．

また，排他制御は threading.Lock クラスのインスタンスを作成し,，acquire() でロックし，release() で開放する．

詳細は https://docs.python.org/ja/2.7/library/threading.html を参照しよう．

```
#!/usr/bin/env python

import time
import threading

# test-thread.py

shared_resource = 0

def task(loop):
    global shared_resource
    for i in range(loop):
        print("Thread {}: i:{}, shared_resource:{}".format(
            threading.current_thread().ident, i, shared_resource))
        shared_resource = shared_resource + 1
        time.sleep(10.0/loop)

thread1 = threading.Thread(target=task, args=([10]))
thread2 = threading.Thread(target=task, args=([20]))
thread1.start()
thread2.start()

thread1.join()
thread2.join()
print("shared_resource {}".format(shared_resource))
```

5 プロセス間通信

プロセス間でのデータ交換を Inter Process Communication(IPC) と呼び，いくつかの方法がある．

5.1 共有メモリ

共有メモリは複数のプロセス間で同じメモリ領域を同時並行的にアクセスしデータを共有する方法である．プロセス内のメモリと同等の速度でアクセスできる一方，２つのプロセスから同時に書き込みを行うことが可能であり，排他機構は用意されておらず，必要に応じて自身で準備する必要がある．

共有メモリで必要になるシステムコールの定義を以下に示す．以下は man ページの記述を要約したものであり，より詳細は man で調べてほしい．

```
#include <sys/mman.h>
#include <sys/stat.h> /* mode 定数用 */
#include <fcntl.h> /* O_* 定数の定義用 */

int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
-lrt でリンクする。

     shm_open() は、POSIX 共有メモリーオブジェクトを新規に作成/オープンしたり、 すでに存在するオブジェクトをオープンしたりする。 POSIX 共有メモリーオブジェクトは、実際には、関係のないプロセスが 共有メモリーの同じ領域を mmap(2) するために使用することができる手段である。 shm_unlink() は、逆の操作、つまり以前に shm_open() で作成されたオブジェクトの削除を行う。

     oflag はビットマスクで、 O_RDONLY と O_RDWR のいずれか一方と、O_CREAT, O_EXCL, O_TRUNK フラグの論理和をとったもの を指定する。

     O_CREAT は存在しない場合、共有メモリーオブジェクトを作成する。 オブジェクトのユーザーとグループの所有権は、 呼び出し元プロセスの対応する実効 ID が使われ、 オブジェクトの許可ビットは mode の下位 9 ビットに基づいて設定される。mode を定義するために使用できるマクロ定数 (群) は open(2) に記載されている。

     成功して完了した場合、 shm_open() は共有メモリーオブジェクトを参照する新しいファイルディスクリプターを返す。
```

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

mmap() は、新しいマッピングを呼び出し元プロセスの仮想アドレス空間に作成する。 新しいマッピングの開始アドレスは addr で指定される。マッピングの長さは length 引き数で指定される。addr が NULL の場合、カーネルがマッピングを作成するアドレスを選択する。 この方法は最も移植性のある新しいマッピングの作成方法である。

ファイルマッピングの内容は、 ファイルディスクリプター fd で参照されるファイル (もしくは他のオブジェクト) のオフセット offset から開始される length バイトのデータで初期化される。

引き数 prot には、マッピングのメモリー保護をどのように行なうかを指定する (ファイルのオープンモードと矛盾してはいけない)。 prot には、PROT_NONE か、PORT_EXEC、PORT_READ、PORT_WRITE の論理和 (OR) をとったものを 指定できる。

fags 引き数により、マッピングに対する更新が同じ領域をマッピングしている 他のプロセスに見えるか、更新がマッピング元のファイルを通じて 伝えられるか、が決定される。この動作は、MAP_SHARED, MAP_PRIVATE の値のいずれか一つだけ (複数は指定できない) を flags に含めることで指定する。

前出の test-fork.c において，プロセス間で情報 (shared_resource) を共有した例が以下になる.

```
/* test-fork-mmap.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* exit */
#include <sys/wait.h> /* wait */

#include <sys/mman.h>
#include <fcntl.h>  // O_RDWR, O_CREAT
#include <string.h> // mmap

typedef struct {
  int shared_resource;
} shared_data_type;

int main () {
  int fd;
  shared_data_type shared_data;
  void *addr;

  fd = shm_open("/NAME", O_CREAT | O_RDWR, 0600); // 0600 = u+rw
  ftruncate(fd, sizeof(shared_data_type));
  addr = mmap(NULL, sizeof(shared_data_type), PROT_WRITE|PROT_READ, MAP_SHARED, fd, 0);
  shared_data.shared_resource = 0;
  memcpy(addr, &shared_data, sizeof(shared_data_type)); // write shared_data to addr(memory)
```

```
  pid_t pid;
  pid = fork();
  if ( pid == 0 ) {
    /* 子プロセス */
    int i, j;
    for (i = 0; i < 10; i++ ) {
      for(j=0;j<10000;j++){
        memcpy(&shared_data, addr, sizeof(shared_data_type)); // write addr to shared_data
        shared_data.shared_resource++;
        memcpy(addr, &shared_data, sizeof(shared_data_type)); // write shared_data to addr
      }
      printf("Process [%d]: i:%d shared:%d\n", pid, i, shared_data.shared_resource);
      sleep(1);
    }
    printf("Process %d: finished, return %d.\n", pid, shared_data.shared_resource);
    exit(0);
  } else if ( pid > 0 ) {
    /* 親プロセス */
    int i, j;
    for (i = 0; i < 20; i++ ) {
      for(j=0;j<10000;j++){
        memcpy(&shared_data, addr, sizeof(shared_data_type)); // write addr to shared_data
        shared_data.shared_resource++;
        memcpy(addr, &shared_data, sizeof(shared_data_type)); // write shared_data to addr
      }
      printf("Process [%d]: i:%d shared:%d\n", pid, i, shared_data.shared_resource);
      usleep(300*1000); // 300 msec
    }
    int status;
    wait(&status);
    printf("Process %d finished, return %d.\n", pid, shared_data.shared_resource);
  }

  //shm_unlink("/NAME");

  return 0;
}
```

コンパイル時には

```
gcc -o test-fork-mmap test-fork-mmap.c -lrt
```

として-lrt をつけることを忘れないようにすること．このプログラムでは排他制御が適切に行われていないことを確認できる．

5.2 セマフォ

セマフォは複数のプロセスが共有する資源にアクセスすることを制御するための変数，または，抽象データ型である．具体的には，ある資源が何個，利用可能かを示す記録であり，その資源を利用する際に数値をインクリメントし，その資源を開放する際にその数値をデクリメントし，使用可能な情報を提供するものである．任意個の資源を扱うセマフォをカウンティングセマフォ，値が 0 と 1 に制限されているセマフォをバイナリセマフォと呼ぶ．後者はミューテックスと同等の機能を

持つが，一般の場合ミューテックスには所有者の概念を持ち，ミューテックスをロックしたプロセスだけがアンロックすることができることに対し，セマフォはそのような制限がない.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);

     shmget () は key 引き数に対応する共有メモリ・セグメントの識別子を返す。
      key の値が IPC_PRIVATE の場合、もしくは semflg に IPC_CREAT が指定さ
     れていて、 key に対応するセマフォ集合が存在しない場合、 nsems 個のセマ
     フォからなる新しい集合が作成される。

     セマフォ集合作成時に、引き数 semflg の下位 9 ビットは、そのセマフォ集
     合の (所有者 (owner)、グループ (group)、 他人 (others) に対する) アク
     セス許可の定義として使用される。 これらのビットは open(2) の引き数
      mode と同じ形式で同じ意味である 。

int semctl(int semid, int semnum, int cmd, ...);

     semctl() は、 semid で指定された System V セマフォ集合 (semaphore set)
      またはセマフォ集合の semnun 番目のセマフォに対して、 cmd で指定された
     制御操作を行なう。

     この関数は、 cmd の値に依存して、3 個または 4 個の引き数を持つ。 引き
     数が 4 個の場合、第 4 引き数の型は union semun である。

     cmd として有効な値は man を参照されたいが SETVAL の場合，集合の semnum 番
     目のセマフォのセマフォ値 (semval) に arg.val の値を設定する。セマフォ
     の値の変更により、他のプロセス内でブロックされている semop(2) コールの
     続行が許可されると、それらのプロセスは起こされる (wake up)。 呼び出し
     たプロセスはそのセマフォ集合に 変更 (書き込み) 許可を持たなければならない。

int semop(int semid, struct sembuf *sops, size_t nsops);

     semop() は semid で指定されたセマフォ集合の選択されたセマフォに対して
     操作を行う。 sops は nsops 個の要素の配列を指し、配列の各要素は個々の
     セマフォに対する操作を示す構造体である。その型は struct sembuf で、次
     のメンバを持つ:

     unsigned short sem_num; /* セマフォ番号 */ short sem_op; /* セマフォ操作 */
     short sem_flg; /* 操作フラグ */

     sops に含まれる操作の集合は、 配列の順序 で、 アトミックに 実行される。
     すなわち、全ての操作が完全に実行されるか、全く実行されないかの どちらかとなる。

     sem_op が正の整数の場合、その値をセマフォの値 (semval) に加算する。 こ
     の操作は必ず実行でき、 スレッドの停止は起こらない。

     sem_op が 0 未満の場合、semval が sem_op の絶対値以上の場合は、操作は
     直ちに実行される。 semval が sem_op の絶対値より小さくい場合、 semval
      が sem_op の絶対値以上になるまでスレッドを停止 (sleep) する。
```

実際の使い方としては以下のようにセマフォを初期化する.

```
union semun {
  int val;
  struct semid_ds *buf;
  ushort * array;
} argument;

int id;
id = semget(12345, 1, 0666 | IPC_CREAT);
argument.val = 1;
semctl(id, 0, SETVAL, argument);

struct sembuf operations[1];
operations[0].sem_num = 0;
operations[0].sem_op = -1;
operations[0].sem_flg = 0;
```

クリティカルセクションで資源をロックする場合は

```
operations[0].sem_op = -1;
semop(id, operations, 1);
```

とし，これをアンロックする場合には

```
operations[0].sem_op = 1;
semop(id, operations, 1);
```

とする．

5.3 ソケット通信

ソケット通信は，異なるマシン上で動いているプロセスと通信する機構である．

ここでは，通常のファイルと同様に，ファイルディスクリプタを介して read,write システムコールを利用してデータの読み書きが可能になる．この場合は open システムコールではなく socket でファイルディスクリプタを生成し，サーバ側は bind で IP アドレスとポート番号を割りあえて，listen で接続待ちを開始し，accept() で読み書き用のファイルディスクリプタを生成する．一方，クライアント側は connect でサーバと接続する．クライアントはソケットディスクリプタをファイルディスクリプタとして利用できる．

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol)
     socket() は通信のための端点 (endpoint) を作成し、ソケットディスクリプター
     (descriptor) を返す。


int  bind(int  s, struct sockaddr *my_addr, socklen_t len);
     socket() で作成したソケット (通信の末端) に，IP アドレスとポート番号を
     割り当てる．s はソケットのディスクリプタ，my_addr は IP やポート番号の
     情報を格納する構造体へのポインタ，len は my_addr(構造体) のサイズ．
```

```
int listen(int s, int backlog)
     listen() はソケットキュー (待ち行列) を作成し，クライアントからの接続要求に応える事
     ができるようにする．その接続可能なクライアントの最大数を backlog で指定する．
     一対一の通信の場合には，backlog は 1 で良い．
     クライアントからの接続要求は，ソケットキューに自動的に入り，次の処理 (accept)
     を待つことになる．

int accept (int s, struct sockaddr_in *addr, int *addrlen)
     accept() は接続要求を受け入れ，その接続用のファイルディスクプリタを返す．
     s はソケットのディスクプリタ，addr はクライアントの情報の受け取りポインタ，
     addrlen はクライアントの情報の長さ．

int connect(int sockfd, const struct sockaddr *addr,
            socklen_t addrlen)
     ソケットが SOCK_STREAM 型であれば、このシステムコールは addr で指定された
     アドレスに結び付けられたソケットに対する接続の作成を試みる．
     クライアントはソケットディスクリプタをファイルディスクリプタと同様に扱い
     read，write システムコールで読み書きが可能である．

struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};
```

プログラムは以下のようにコンパイルする．

```
server: server.c
        gcc -o server server.c

client: client.c
        gcc -o client client.c
```

サーバは./server として実行するが，クライアントは

```
./client 127.0.0.1
```

として第一引数にサーバの IP アドレスを指定し実行する．127.0.0.1 は自分自身の IP アドレスを指す特殊な数値になっている．

また，プログラムが不正終了した後，あるいは，サーバがすでに立ち上がっている状態で，にサーバを立ち上げると

```
$ ./server
server: bind: Address already in use
```

と表示される．この場合はポート番号を他の値に変更するか，30 秒ほど待つとポート番号が再度利用できるようになる．

```
// server.cpp

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h> // sockaddr_in

int main ()
{
  int ip_port = 1024;

  // ソケットの生成
  int s;
  if ((s = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
      perror ("server: socket");
      exit (1);
  }

  struct sockaddr_in sin;
  memset ((char *) &sin, 0, sizeof (struct sockaddr));
  sin.sin_family = AF_INET; // アドレスの型の指定
  sin.sin_port = htons (ip_port); // ポート番号
  sin.sin_addr.s_addr = htonl (INADDR_ANY); // 待ち受けの IP アドレスの設定

  // ソケットにパラメータを与える
  if ((bind (s, (struct sockaddr *) &sin, sizeof (sin))) < 0) {
      perror ("server: bind");
      exit (1);
  }

  // クライアントの接続を待つ
  if ((listen (s, 10)) < 0) {
      perror ("server: listen");
      exit (1);
  }

  int ns;
  printf ("waiting for connection\n");

  // クライアントからの接続をまつ
  if ((ns = accept (s, NULL, 0)) < 0) {
    perror ("server: accept");
    exit (1);
  }
  printf ("Connected : %d\n", ns);

  char buf[256];
  int size;

  // データを受信する
  size = read (ns, buf, 255);
  buf[size] = 0;
  printf("recv (%d) [%s]\n", size, buf);

  // データを送信する
  char buf2[256];
  sprintf(buf2, "Received '%s'!", buf);
  size = write (ns, buf2, strlen(buf2));
  printf ("send (%d) [%s]\n", size, buf2);

  close(s);

  exit (0);
}
```

```
// client.cpp

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h> // sockaddr_in
#include <arpa/inet.h> // ip_addr

int main (int argc, char *argv[])
{
  if (argc == 1)
    {
      printf ("usage: client <server IP address>\n");
      exit (1);
    }

  char* ip_addr = argv[1];
  int ip_port = 1024;

  struct sockaddr_in sin;
  int s;

  // ソケットを生成する
  if ((s = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
      perror ("client: socket");
      exit (1);
  }

  // サーバの情報を与える
  sin.sin_family = AF_INET; // アドレスの型の指定
  sin.sin_port = htons (ip_port); // ポート番号
  sin.sin_addr.s_addr = inet_addr (ip_addr); // 宛先のアドレスの指定 IP アドレスの文字列を変換

  // サーバに接続する
  if ((connect (s, (struct sockaddr *) &sin, sizeof (sin))) < 0) {
      perror ("client: connect");
      exit (1);
  }
  printf ("established connection\n");

  char buf[256];
  int size;
  // データを送信する
  strncpy(buf, "Hello World", 12 /* strlen("Hello World") */);
  size = write (s, buf, strlen(buf));
  printf ("send (%d) [%s]\n", size, buf);

  // データを受信する
  size = read (s, buf, 255);
  buf[size] = 0;
  printf("recv (%d) [%s]\n", size, buf);

  close(s);
  exit (0);
}
```

宿題

提出先：ITC-LMS を用いて提出すること

提出内容：以下の問題の実行結果の画面をキャプチャしファイル名は「問題番号.png」とし，また講義中にでてきたキーワードについて知らなかったもの, 興味のあるものを調べ「学籍番号.txt」としててアップロードすること．テキストファイルはワードファイルなどだと確認出来ないことがあるため，emacs/vi 等のテキストエディタを使って書こう．プログラムが長くなりキャプチャ画面に入り切らなくなってきたらプログラムファイルと実行結果を「問題番号.txt」にまとめてアップロードしてよい．
画像で提出する場合は，各自のマシンの Mac アドレスが分かるようにすること．例えば画面中に ifconfig というコマンドを打ち込んだターミナルを表示すればよい．
ITC-LMS にアップロードする際には講義・宿題の感想を必ずコメントに記すこと．また授業中に質問した者はその旨を記すこと．質問は成績評価時の加点対象となる．

キーワード：Lowlatency カーネルとリアルタイム Linux，POSIX, 並列処理と並行処理，スレッドセーフ

1. test-thread2.c をクリティカルセクションのアトミック性を確保するために排他制御を追加し，期待通りの振る舞いをするよう変更せよ．排他制御でロックするクリティカルセクションの領域（行数）は最小限になるように注意せよ．
   ヒント：mutex オブジェクトは全てのスレッドから共有できるよう大域変数として以下のように定義する．

   ```
   pthread_mutex_t mutex;
   ```

   また，その初期化もスレッド関数の外側（main 文中が望ましい）で以下のように初期化する．ポインタを渡す点に注意する．

   ```
   pthread_mutex_init(&mutex, NULL);
   ```

   排他制御はスレッド関数内で以下のようにしてロック，アンロックを呼び出す．

   ```
   pthread_mutex_lock(&mutex);
   pthread_mutex_unlock(&mutex);
   ```

2. test-thread2.c のプログラムに SIGINT を受け取るシグナルハンドラを追加し，C-c また、このプロセス番号に対して SIGINT シグナルをコマンドラインから送った場合に現在の `shared_resource` が表示されるようにせよ．

3. test-thread.py のプログラムに対して，排他制御がなされていないことが確認できるよう，以下のコードを追加してみよ．また，`threading.Lock()` を用いて排他制御をかけてみよ．

   ```
   # shared_resource = shared_resource + 1
   for j in range(10000):
       tmp = shared_resource
       tmp = tmp + 1
       time.sleep(1/1000000.0)
       shared_resource = tmp
   ```

A コルーチン

サブルーチンが関数のエントリからリターンまでを一つの処理単位とするのに対し、ジェネレータ、コルーチンはいったん処理を中断した後、その続きから処理を再開できるものである．特に再開時に新しい情報を渡す手段を持っているものをコルーチンと呼び、持っていない場合はジェネレータと呼ぶこともある．以下に Python の yeild を用いた例を示す．

```
import time

def loop(n):
    name = "loop"
    for i in range(n):
        print("{} {}".format(name, i))
        yield i
        time.sleep(1/n)

l1 = loop(10)
l2 = loop(10)
for i in range(10):
    next(l1)  # l1.next() for python2
    next(l2)  # l2.next() for python2
```

```
def loop(n):
    name = "loop"
    for i in range(n):
        print("{} {}"
                .format(name, i))
        name = yield i
        time.sleep(1)

l1 = loop(10)
l2 = loop(10)
next(l1)  # l1.next() for python2
next(l2)  # l2.next() for python2
for i in range(9):
    l1.send("name1")
    l2.send("name2")
```

B 継続 (continuation)

継続とは「次に行われる計算」

```
(define (foo) (display "foo"))
(define (bar) (display "bar"))
(define (baz) (display "baz"))
(define (test) (foo) (bar) (baz))
```

とあると、(test) は (foo), (bar), (baz) を順番に呼び出す．foo を呼び出した後の「継続」は (bar),(baz) になる．

```
(* (+ 1 2) (- 10 5))
```

の場合は (+ 1 2) の評価が終わった段階の継続は、この評価の結果を a とすると、(* a (- 10 5)) となる．また、2 が評価された時点の継続は、この結果を a(=2) とすると、(* (+ 1 a) (- 10 5)) となる．

call/cc を用いることで、この継続を取り出すことができる．

```
(call/cc (lambda (cc) [式．cc は call/cc が呼ばれた時点での「継続」が入る]))
```

であるから、上記の式は

```
(* (+ 1 (call/cc (lambda (cc) 2))) (- 10 5))
```

とかける．継続は保存することができるので、

```
gosh> (define *save* #f)
*save*
gosh> (* (+ 1 (call/cc (lambda (cc) (set! *save* cc) 2))) (- 10 5))
15
gosh> (*save* 2)
15
gosh> (*save* 10)
55
```

となる．以下のような lambda 式で表される関数が継続として*save*に保存されていると考えて良い．ただし、トップレベル以外で呼ばれた場合は、その時点でトップレベルに戻るという振る舞いをする．

```
gosh> ((lambda (a) (* (+ 1 a) (- 10 5))) 2)
15
gosh> ((lambda (a) (* (+ 1 a) (- 10 5))) 10)
55
```

```
gosh> (*save* 5)
30
gosh> (/ (*save* 5) 0)
30
```

以下に Scheme の継続を用いたコルーチンの例を示す．

```
(define func0 (lambda ()
                (do ((i 0 (+ i 1)))
                    ((>= i 10) #f)
                  (print i)
                  (pause)
                  )))

(define func1 (lambda ()
                (do ((i 10 (+ i 1)))
                    ((>= i 20) #f)
                  (print i)
                  (pause)
                  )))
```

```
(use util.queue)
(define *queue* (make-queue))
(enqueue! *queue* func0)
(enqueue! *queue* func1)

(define (start)
  (if (not (queue-empty? *queue*))
      ((dequeue! *queue*))
      ;;(let ((f (dequeue! *queue*))) (f))
      ))
(define (pause)
  (call/cc (lambda (c)
             (enqueue! *queue* c)
             (start))))
(start)
```

参考文献

http://www.geocities.jp/m_hiroi/func/abcscm20.html,

https://practical-scheme.net/wiliki/wiliki.cgi?Scheme:使いたい人のための継続入門

http://www.shido.info/lisp/callcc.html