データ収集システムの開発

先端エレクトロニクスDAQセミナー
データ収集システム入門

2011年7月28日
藤井啓文


```
#include <fstream>

int main()
{
        static const int buflength = 1024;

        char buf[buflength];

        std::ifstream is( "mydevice", std::ios::binary );
        std::ofstream os( "myexp.dat", std::ios::binary );

        while (is && os)
        {
                is.read( buf, buflength );
                if ( is.gcount() )
                        os.write( buf, is.gcount() );
        }
        return 0;
}
```

超お気楽DAQシステム


```
int main()
{
        while( read(…) )
                write(…);
}
```

データ収集システムの役割

- 計測したデータや観測したデータを情報として後世に伝える（1秒後の私も後世）。
- データではなく情報（人間にとって意味のあるもの）として伝えることが大事。
- 計算機で後世に伝える手段は半恒久的媒体に記録すること。
- メディアレコーダーと酷似。

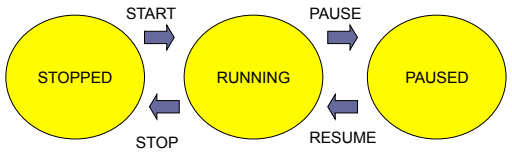
データ収集システムの状態

- データ収集システムを状態マシンとしてとらえると、基本的に３つの状態
  - 停止状態
  - 走行状態
  - 一時停止状態
- 状態遷移を引き起こす操作
  - START 停止状態から走行状態へ
  - STOP 走行状態から停止状態へ
  - PAUSE 走行状態から一時停止状態へ
  - RESUME 一時停止状態から走行状態へ

超お気楽DAQシステム

- 装置からデータを読み、半恒久媒体へ記録するプログラムのみ。
  - START は OS からのプログラムの起動で対応。
  - STOP は OS のプログラム終了で対応。
  - PAUSE は装置の信号の送出を止めることで対応。
  - RESUME は装置の信号の再送出で対応。

データ収集システムの状態



何のために

- メディアレコーダの場合は後から見たい、繰り返し見たい。
- DAQの場合は後から解析したい（解析に時間がかかる）。繰り返し解析したい（パラメータを変える、別の仮説を当てはめるなど）。

何を伝える（記録する）か

- 情報伝達の基本は５W1H
  WHO、WHAT、WHEN、WHERE、WHY、HOW
  これらと測定データ、観測データが一体となって情報となる。
- 管理上、理想的には一つのファイルにまとまっているのが望ましいが必ずしも同じファイル、同じ媒体とは限らない（データファイルとログブック、ログファイルとして RUN# などで関連付けるとか）。

データの内部表現

- 通常、計測器からのデジタルデータは整数。
- 扱える範囲に注意。現在の C/C++ 標準規格では保証範囲は32ビットまで。
- 符号に注意。特に負の値。(処理系による。必ずしも２の補数ではない。)
- C/C++標準規格で取り扱うなら、符号無し整数で扱う方がよい。（２のｎ乗を法とする算術演算であることが保証されている）。

32ビットを超える整数

- C++ なら Int64 などのクラスを作る（借りてくる）手もある。
- もっと安易に double にする手もある。浮動小数点数の仮数部分の精度がある。（IEEE-754を使っていれば52ビットある）。

停止条件を与えたい

- 停止条件の例
  - 一定量のデータが溜まったらデータ収集システムを停止する。
  - 一定時間観測したらデータ収集システムを停止する。
- ハードウェアに任せる手もある。
- プログラムでやろうとすると、途端にプログラム設計上の問題が出てくる。

停止条件を与えるために

- パラメータの入力（UI）をどうするか？
- 不正パラメータの取り扱いをどうするか？
- 与えた条件は記録しておきたい
  - データファイルに入れるのか？入れるとしたらデータとの区別（ファイル構造）は？
  - 別ファイルにするのか？別ファイルにするとしたらデータファイルとの関連付けは？（データファイル名を記録するか？。データベースシステムを使うか？）
- 「いつ誰が何のために」も入力・記録させたい
  - 本番データ収集以外にも同一プログラムでテストランをすることがあるとか。


```
#include <iostream>
#include <sstream>
int usage(char* progname)
{
	std::cerr
		<< "Usage: " << progname << " maxevents" << std::endl;
	return (-1);
}
int main(int argc, char* argv[])
{
	if (argc != 2)
		return usage( argv[0] );
	std::stringstream ss( argv[1] );

	int maxevents;
	if (!(ss >> maxevents))
	{
		std::cerr << "parameter error (must be integer)" << std::endl;
		return (-2);
	}
	if (maxevents <= 0)
	{
		std::cerr << "parameter must be positive interger" << std::endl;
		return (-3);
	}
	std::cout << maxevents << std::endl;
	return (0);
}
```

入力インターフェース

- 今時のシステムでは GUI、でもコマンドラインでも十分なことも。
  - GUI は後から外付けで実装することも可能なように設計しておく。
- 多くの C/C++ 言語処理系では、コマンドラインは main() 関数の引数。
  - 言語の標準規格で決まっているわけではないことに注意。特に非ASCII文字列の取り扱いなど要注意（名前の入力などで起こる）。下手をすると異常終了を引き起こす。

文字コード（「誰が」を書く）

- 問題を避けたいなら ASCII コードに限定。
  - でもそんなこと気にせず入力するユーザがいるだろう。対策必要。
- 可能なら、できるだけ UNICODE を使う。
  - 多くの国の文字を統一的に使える。
- 可能なら、UTF-8 でエンコードする。
  - Web（W3C）の標準エンコード方式。
  - 可変サイズ（文字数とオクテット数が一致しない）注意。多くの漢字は3オクテット。
  - いわゆる BOM （バイト順マーク：FF BB BF）の付与が問題を引き起こすことがある（このあたり議論うるさい）。

時刻情報（「いつ」を得る）

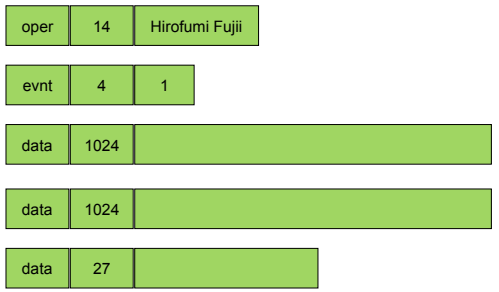
- C/C++ では標準ライブラリがある。
  - ヘッダは <time.h> (C), <ctime> (C++)
  - 時刻は time() 関数（内部表現での値）
  - UTC 時刻を得るのは gmtime()　ローカル時刻を得るのは localtime() 関数
  - 文字表示にするには asctime() 関数
  - 時刻差は difftime() 関数（double で秒単位）
  - プロセス経過時間は clock() 関数 (CLOCKS_PER_SEC単位で秒)

```
#include <iostream>
#include <ctime>
int main()
{
        time_t tcur = time( 0 );
        std::cout << asctime( gmtime( &tcur ) );
}
```


ファイルに構造を持たせる

- 一つのファイルに型の異なる情報を入れる。
  - パラメータと計測データを一つのファイルに。
- CHUNK構造とかTAG構造と言われる方法。
  - 長さ（通常４オクテット＝３２ビット）＋識別子（４ASCII文字）＋データ。（長さと識別子の順は異なる形式もある）。
  - MP4ファイルフォーマット（ISO/IEC 14496-14）やPNGファイルなど画像やマルチメディア情報を格納する技法として多く用いられている。

複数の型を一つのファイルに



正常に収集できているかを知る

- データ収集システムとしては正常でも、実験や観測としては異常ということがあり得る。
  - 計測器の電圧設定を間違えた
  - 計測器が雑音だらけになってしまった
- 異常は、できるだけ早く検出して処置したい。長時間の放置はリスク大。
- 収集したデータの情報を常時表示する（モニタ）必要性あり。

モニタの役割

- このままランを継続してよいか否かの判断情報をユーザに与える。
- 人に伝えることが目的なので、ユーザインターフェース（UI）が重要。
- 異常（上限、下限とか）設定ができるようにして異常検知したら騒ぎまくる（ベルを鳴らす、画面をチカチカさせる）のがよい。
- でも UI は（人が絡むので）大変。
- 一定時間毎にモニタ情報をファイルに書き出すだけでもメリット大（ちょっとの努力で大きなメリット）。

複数入力

- 一般にデータ源は複数。データの発生順は決められないことが多い。
- 順に読んでいくと、途中で遅れたデータ源があると、そこで待たされる。
- 現時点での解法としてはマルチスレッドがよく用いられる。

```
for (I = 0; I < maxevents; i++)
{
        read(dev1,buf1…);
        read(dev2,buf2…);
        read(dev3,buf3…);
                :
        write(outfile,buf1…);
                :
}
```

もし dev1 で待たされると、dev2、dev3 がすでに読める状態になっていても待たされる。

マルチスレッド

- 一つのプログラムで複数のコードを並列実行。
- 命令実行は独立（CPU資源はスレッド毎に確保）。その他の資源（メモリ、ファイル等）は共通。マルチプロセスに比べ起動が早く軽い。
- CPU資源以外は共通なので、自分で管理を行う必要性。

マルチスレッドCインターフェース

- POSIX C では
  - 生成 pthread_create()
    これにより pthread_t 型のオブジェクト
  - 終了を待つ pthread_join()

排他制御

- 他のスレッドに資源を使わせない仕組み。
- 通常は mutex を使う。
- 特定資源に結びついた mutex を共通領域に確保。
- その資源を使おうとするスレッドは、mutex を取得してから使う（一般に取得できるまで待たされる）。
- その資源を使い終えたら mutex を解放。

同期

- スレッド間で資源が準備できたことを伝える仕組み。
- 読み手スレッドと書き手スレッドの同期をとるのが一般的（リングバッファなど）。
- 通常は (counting) semaphore を使う。
- 読み手は 読む前に semaphore を取得（0だと待たされる）。書き手は書き終わったら semaphore を増加させる。

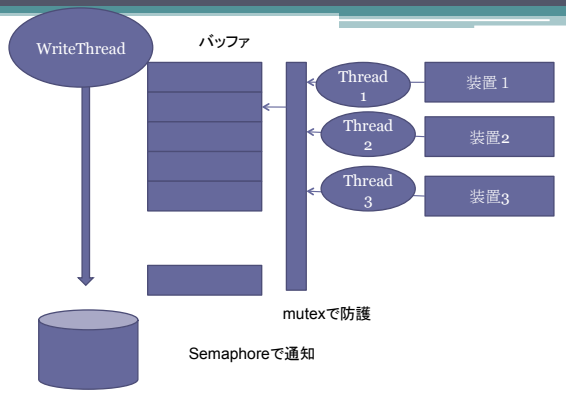
```
void* thead1(void* arg)
{
        while(read(dev1,buf…))
                write(outfile,buf…)
}
void* thread2(void* arg)
{
        while(read(dev2,buf…))
                write(outfile,buf…)
}
Int main()
{
        pthread_create(..thread1..);
        pthread_create(..thread2..);
```


semaphore

- POSIX C では sem_t 型オブジェクト
  ヘッダは <semaphore.h>
- 最小限必要な操作
  - 生成（初期化） sem_init()
  - 削除 sem_destroy()
  - 確保 sem_wait()
  - 通知 sem_post()
- 監視可能な操作
  - 監視付き確保 sem_trywait()

mutex

- POSIX C では pthread_mutex_t 型オブジェクト
- 最小限必要な操作
  - 生成（初期化） pthread_mutex_init()
  - 削除 pthread_mutex_destroy()
  - 確保 pthread_mutex_lock()
  - 解放 pthread_mutex_unlock()
- 監視可能な操作
  - 確保の試行 pthread_mutex_trylock()



多重同期

- 複数の装置からのデータを同期がとれるように一つのファイル（ストリーム）にまとめる。
- パケット化して、個々のパケットにタイムスタンプ（共通クロックやトリガー番号など）、をつける。
- 典型例は地デジのストリーム（MPEG2-TS）。一つのストリームに複数の動画（フルセグ、ワンセグ混在）、複数の音声ストリームや字幕ストリームを含む。

まとめ

- DAQシステムは後から何度でも解析できるように、情報として記録するシステム。どのような情報をどのように記録しておくか十分に考察する。
- 正常に働いていることを常時示す。そのためには正常を示すパラメータは何かを考察する。
- いつでも、どこでも必要な情報を得られるように可能な限り標準化された規格などを使う。標準規格を使えばテストツールなども揃っていることが多い。
- ソフトウェアにすべての解を見つけようとせず、広く柔軟に解をさがす。