C 言語におけるファイル入出力の高速化

東京大学情報基盤センター
黒田 久泰

不連続なメモリ上のデータをファイルに保存する場合、内部バッファサイズを大きくすると実行時間が短縮できます。また、メモリ上に連続して配置されている大規模なデータをファイルに保存する場合には、できるだけ大きなデータサイズでファイル入出力を行うことで実行時間が短縮できます。ここでは、これらの方法や性能について紹介します。

1. 内部バッファサイズの変更方法

高水準入出力関数 fopen、fread、fwrite、fclose では内部バッファにデータを貯めておき、内部バッファが空になるか一杯になるとまとめてファイル入出力を行います。通常、この内部バッファサイズはインクルードファイル /usr/include/stdio.h の BUFSIZ で定義されている値になります。例えば、SR8000/MPP では 64KB、SR11000/J1 では 4KB になっています。

(サンプルプログラム main.c)

```
#include <stdio.h>
#include <sys/time.h>
#define SIZE 256*1024*1024
int a[SIZE][2];

double getETime()
{
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return tv.tv_sec + (double)tv.tv_usec*1e-6;
}

int main(int argc,char *argv[])
{
  int i;
  double st,en;
  FILE *fp;

  st=getETime();
  fp=fopen("a.dat","w");
  for(i=0;i<SIZE;i++){
     fwrite(&a[i][0],sizeof(int),1,fp);
  }
  fclose(fp);
  en=getETime();

  printf("Elapsed Time=%.6f¥n",en-st);
  return 0;
}
```

このプログラムは配列 a[SIZE][2]の半分の要素 a[0][0], a[1][0], a[2][0], …, a[SIZE-1][0]だけを抜き出してファイルに出力するプログラムです（出力されるファイルサイズは 1GB)。getETime 関数は gettimeofday システムコールの値を double 型の秒数に変換して返す関数で、en-st はファイル出力にかかった経過時間(秒)を表します。

このプログラムを実行すると、下記のようになります。

```
(SR8000/MPP)
 % cc -64 -Os -noparallel +Op main.c
 % ./a.out                              (←実際はバッチジョブ上で実行)
 Elapsed Time=182.191025
(SR11000/J1)
 % cc -64 -Os -noparallel +Op main.c
 % ./a.out                              (←実際はバッチジョブ上で実行)
 Elapsed Time=237.864731
```

SR8000/MPP では約 182 秒、SR11000/J1 では約 238 秒かかりました。このサンプルプログラムでは、内部バッファサイズを変更することで高速化させることが可能です。バッファサイズを変更するには、setvbuf 関数を使用します。

【書式】

```
#include <stdio.h>
int setvbuf(FILE *fp, char *buffer, int mode, size_t size);
```

【説明】

fp で示されるファイルに対して、自動的に割り当てられる入出力バッファの代わりに buffer で指定した領域を入出力バッファとして利用することを指定する。size には新しく割り当てる入出力バッファのサイズを指定する。buffer の値が NULL の場合には malloc を使って自動的に size 分のメモリを確保する。mode には次の 3 つの内のどれかを指定する。

_IOFBF　入出力を完全バッファリング（バッファを埋め尽くすと出力を行う）
_IOLBF　入出力を行バッファリング（バッファを埋め尽くしたときと改行コードが来たときに出力を行う）
_IONBF　バッファリングなし（buffer と size の値は無視される）

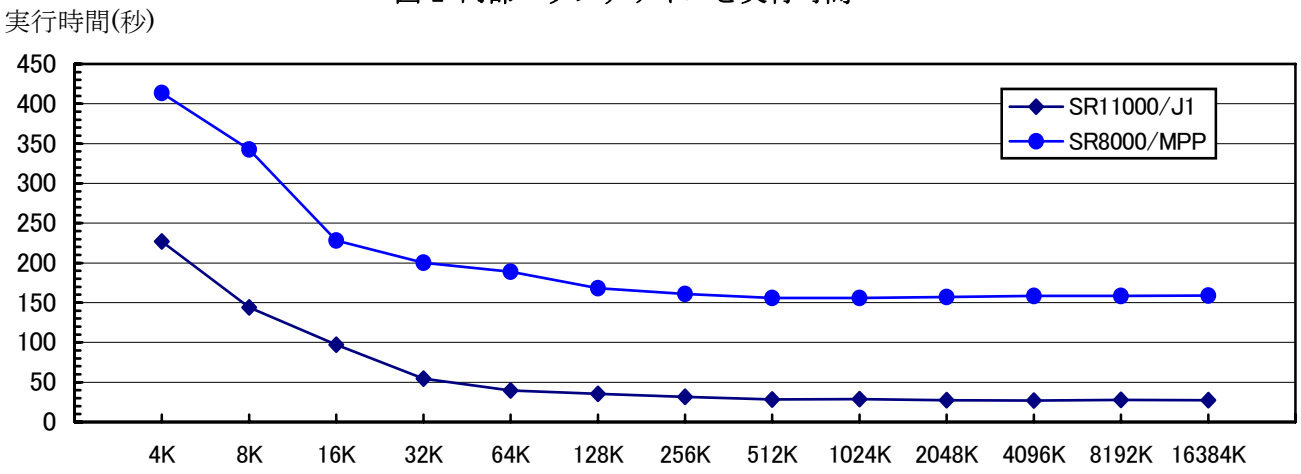
例えば入出力バッファサイズを 512KB にする場合には、下記のようにコードを追加します。

（内部バッファサイズの変更を行うコードの追加）

```
fp=fopen("a.dat","w");
setvbuf(fp,NULL,_IOFBF,512*1024);      ←この 1 行を追加
for(i=0;i<SIZE;i++){
   fwrite(&a[i][0],sizeof(int),1,fp);
}
fclose(fp);
```

fopen 関数でファイルをオープンした後に、setvbuf 関数を呼び出すことに注意してください。
内部バッファサイズを 4KB から 32MB まで変えたときの実行時間は図 1 のようになります。

図 1 内部バッファサイズと実行時間



SR8000/MPP では入出力バッファサイズを 512KB にすると実行時間は約 156 秒になり標準の 64KB のときと比べて約 1.17 倍の速度向上となります。一方、SR11000/J1 では標準の入出力バッファサイズが 4KB と小さいため、512KB にすると実行時間は約 28 秒になり約 8.4 倍の速度向上となります。

2. 低水準入出力関数を用いたファイル入出力

低水準入出力関数である open、read、write、close を使ってファイル入出力を行う場合、内部バッファを使わずに直接ディスクに対してファイルの読み書きを行うことができます。これらはシステムコールと呼ばれ、直接、オペレーティングシステム（OS）によって処理されます。

【書式】

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags, mode_t mode);
```

【説明】

pathname にはファイルのパス名、flags にはファイル状態フラグ、mode にはファイルのアクセス属性を指定してファイルをオープンする。flags の値としてよく使われるものは次のものである。ただし、O_RDONLY、O_WRONLY、O_RDWR のどれか１つは入っていなければならない。

O_RDONLY	読み込み専用
O_WRONLY	書き込み専用
O_RDWR	読み書き両用
O_CREAT	ファイルが存在しなかった場合作成する
O_APPEND	追加モードでオープンする（ファイル・ポインタをファイルの最後に移動）
O_TRUNC	ファイルが既に存在する場合に書き込みモードでオープンされている場合、ファイルの長さを 0 に切り詰める

mode には下記のシンボル定数を指定する（新しくファイルを作成するときのみ使用される）。

S_IRUSR	ファイル作成者に読み込みの許可がある
S_IWUSR	ファイル作成者に書き込みの許可がある
S_IXUSR	ファイル作成者に実行の許可がある

ファイル作成者と同一グループの者に許可を与える場合にはそれぞれ S_IRGRP、S_IWGRP、S_IXGRP を指定し、他人に許可を与える場合にはそれぞれ S_IROTH、S_IWOTH、S_IXOTH を指定する。

【書式】

```
#include <unistd.h>
ssize_t read(int filedes, void *buffer, size_t nbytes);
ssize_t write(int filedes, void *buffer, size_t nbytes);
```

【説明】

read ではデータの読み込み、write ではデータの書き込みを行う。filedes にはファイルディスクリプタ、buffer にはデータの先頭アドレス、nbytes にはデータのバイト数を指定する。

【書式】

```
#include <unistd.h>
int close(int filedes);
```

【説明】

filedes で示されるファイルディスクリプタをクローズする。

低水準入出力関数を利用すればいつでも速くなるわけではありません。具体的にその例を示します。

次の 2 つのプログラムは、大きさ 100 万の配列 a の値を 1 要素（＝4 バイト）ずつファイルに出力するプログラムです。左のプログラムは高水準入出力関数 fopen、fwrite、fclose で書かれており、右のプログラムでは低水準入出力関数 open、write、close で書かれています。

(fwrite を使った場合)

```
#include <stdio.h>
#define SIZE 1000000
int a[SIZE];

int main(int argc,char *argv[])
{
  int i;
  FILE *fp;

  fp=fopen("a.dat","w");
  for(i=0;i<SIZE;i++){
    fwrite(&a[i],sizeof(int),1,fp);
  }
  fclose(fp);
  return 0;
}
```

(write を使った場合)

```
#include <sys/types.h>
#include <sys/mode.h>
#include <fcntl.h>
#define SIZE 1000000
int a[SIZE];

int main(int argc,char *argv[])
{
  int i;
  int file;

  file=open("a.dat",O_WRONLY|O_CREAT,S_IREAD|S_IWRITE);
  for(i=0;i<SIZE;i++){
    write(file,&a[i],sizeof(int));
  }
  close(file);
  return 0;
}
```

どちらも「cc -64 -Os -noparallel +Op」でコンパイルして実行したところ、下記のような実行時間になりました。

	fwrite を使った場合	write を使った場合
SR8000/MPP	1.21 秒	195.68 秒
SR11000/J1	0.29 秒	166.98 秒

このように write を使って小さいサイズの書き込みを何度も行うと、極端に遅くなってしまいます。低水準入出力関数を使う場合には、できるだけ大きなサイズ単位でファイル入出力を行うように心がける必要があります。

3. ファイルシステム para-io の利用

単一の大容量ファイルの入出力を行う場合には、ファイルシステム para-io を利用すると実行時間が短縮されます。ここではファイルシステム para-io を利用して大きなサイズのファイルを出力したときの性能を紹介します。

性能測定プログラムには下記のプログラムを利用しました。配列 a は double 型でデータサイズは 8GB になります。この 8GB のデータを複数回に分けてファイル出力を行います。ここでは低水準入出力関数を利用したプログラムを示します。

(性能測定プログラム)

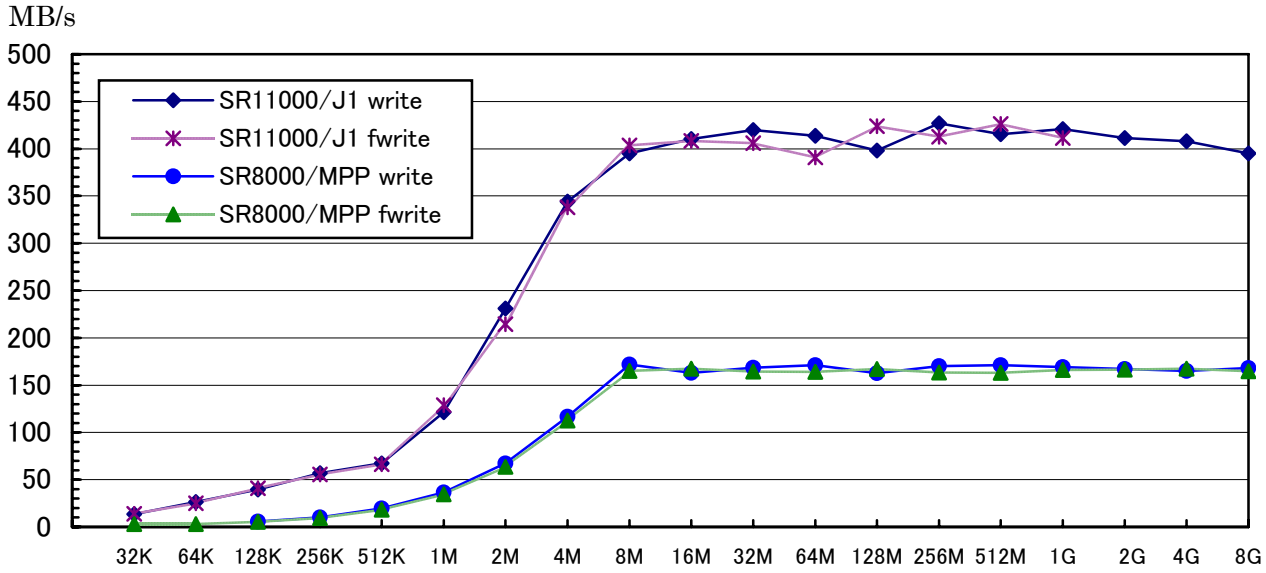
```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/mode.h>
#include <fcntl.h>
#include <unistd.h>
#define SIZE 1024*1024*1024
double a[SIZE];

double getETime()
{
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return tv.tv_sec + (double)tv.tv_usec*1e-6;
}

int main(int argc,char *argv[])
{
   long long int i,bsize;
   double st,en;
   int file;   ←――――――――――――――――――――――――――――――――――――――――――――――――――――――――①

   for(bsize=4096;bsize<=SIZE;bsize*=2){
      unlink("/para-io/******/a.dat");
      st=getETime();
      file=open("/para-io/******/a.dat",O_WRONLY|O_CREAT,S_IREAD|S_IWRITE);  ←――②
      for(i=0;i<SIZE/bsize;i++){
         write(file,&a[bsize*i],sizeof(double)*bsize);   ←――――――――――――――③
      }
      close(file);   ←――――――――――――――――――――――――――――――――――――――――――――――――――――④
      en=getETime();
      printf("Size=%lld Elapsed Time=%.6f Rate=%.3fMB/s¥n",
             bsize*8,en-st,(double)SIZE*8/1024/1024/(en-st));
   }
   return 0;
}                                          ※ ****** の部分にはユーザーID を記述すること
```

高水準入出力関数を利用したプログラムに変更するには、①から④までをそれぞれ下記のように修正します。

① FILE *fp;
② fp=fopen("/para-io/******/a.dat","w");
③ fwrite(&a[bsize*i],sizeof(double),bsize,fp);
④ fclose(fp);

１回に書き込むサイズを 32KB から 8GB まで増やしていったときのディスク書き込み速度のグラフを図２に示します。

図２ １回に書き込むサイズとディスク書き込み速度



低水準入出力関数 write を使ったプログラムと高水準入力関数 fwrite を使ったプログラムでは、実行時間にほとんど差がありませんでした。これは、内部バッファサイズと同じサイズ以上のデータを fwrite で書き込もうとした場合に、内部バッファ経由ではなく、直接、ディスクに書き込む処理を行っているからだと考えられます。そのため、低水準入出力関数を使用しなくても、高水準入出力関数を使えば十分な性能が得られることになります。

ファイルシステム para-io を使う場合、1 回に書き込むサイズを 8MB 以上になるようにすれば、SR8000/MPP では約 170MB/s、SR11000/J1 では約 420MB/s という転送速度になります。

補足事項

1. SR11000/J1 では、仕様により fwrite で 2GB 以上のデータを一度に書き込もうとすると失敗します。

   ```
   fwrite(a,sizeof(double),bsize,fp);
   ```

   のように書かれている部分を、

   ```
   write(fileno(fp),a,sizeof(double)*bsize);
   ```

   のように write に書き換えると、2GB 以上のデータも書き込むことができるようになります。
   fileno はファイルハンドルに対応するファイルディスクリプタを返すマクロ関数です。

2. fwrite で内部バッファサイズを超えるデータをファイルに出力する場合、内部バッファサイズ単位毎にデータを内部バッファにコピーしてファイル出力を行うか、あるいは、内部バッファを経由せずに直接データ全体を出力するかはシステムに依存します。これらはシステムコール追跡コマンドで open システムコールの呼び出し部分を解析することで調べることができます。システムコール追跡コマンドは、SR11000/J1 では truss、FreeBSD であれば ktrace、Linux であれば strace というコマンド名になります。