情報科学演習 C 課題 2 レポート

担当教員：小島 英春
提出者：小山 亮
学籍番号：09B15028
e-mail：u745409b@ecs.osaka-u.ac.jp

提出年月日：平成 29 年 7 月 3 日

1 課題内容

3-1 課題 3 2.2.1 節の file-counter プログラムを以下の動作をみたすように改変せよ。

- counter ファイルの内容が最終的に 4 になる
- 実行ごとに結果が変化しない
- file-counter プログラムが単体で動作する

3-2-1 課題 3 4.1 節の pipe プログラムを拡張し、双方向パイプのプログラム two-way-pipe を作成せよ。

3-2-2 課題 3 に添付された mergesort.c を拡張し、並列版マージソートを作成せよ。

3-3-1 alarm() を使用せずに、同等の機能を実現する myalarm() をマルチプロセスを用いて実装せよ。

3-3-2 課題 3-3-1 で作成した myalarm() を利用して、課題 2 で作成した simple-talk-client プログラムにタイムアウト機能を実装せよ。

2 課題 3-1

課題 3 2.2.1 節の file-counter プログラムは子プロセスがファイルの内容を 1 ずつ加算していくプログラムであるが、子プロセスの生成が排他的でないために、ほかのプロセスが動いている間に、違うプロセスが動作し、期待する結果と異なる場合がある。また、プロセスが生成されるタイミングは予測できないので、実行ごとに結果が異なる。

2.1 設計の説明

問題点を解決するために、セマフォを用いて子プロセスを排他的に動作させ、複数のプロセスが同時に目的のファイルにアクセスしないようにした。セマフォを獲得する部分に当たるコードを下に示す。

```
if ((key = ftok(".", 1)) == -1){
  fprintf(stderr,"ftok path does not exist.\n");
  exit(1);
}
if ((sid=semget(key, 1, 0666 | IPC_CREAT)) == -1) {
  perror("semget error.");
  exit(1);
}
```

ftok() を用いて key を作成した後、 semget() でセマフォを獲得する。semget() の第 3 引数には 0666 を指定し、全ユーザが読み書き可能であることを意味している。プロセスを 4 つ生成するループの部分を下に示す。このん部分がクリティカルセクションとなっている。

```
        for (i=0; i<NUMPROCS; i++) {
                sb.sem_op  = -1;
                if (semop(sid, &sb, 1) == -1) {
                        perror("sem_wait semop error.");
                        exit(1);
                }
                if ((pid=fork())== -1) {
                        perror("fork failed.");
                        exit(1);
                }
                if (pid == 0) { /* Child process */
                        count = count1(ct);
                        printf("count = %d\n", count);

                        sb.sem_op  = 1;
                        if (semop(sid, &sb, 1) == -1) {
                                perror("sem_signal semop error.");
                                exit(1);
                        }
                        exit(0);
                }
        }
```

まず親プロセスはセマフォの値を -1 にして待ち状態に入る。子プロセスを生成して、子プロセスでの処理を終えると、セマフォの値を 1 にして、親プロセスを続行させる。こうすることで、同時に 2 つの子プロセスが動作することはなくなる。

2.2 実行結果

```
r-koyama@exp007:~/enshuuC/kadai3$ ./file-counter
count = 1
count = 2
count = 3
count = 4
```

期待する結果が出ていることがわかる。何回実行しても結果は変わらなかった。

2.3 考察等

子プロセスの終了を待たずに、ほかの子プロセスを生成する処理を行うと、並列して子プロセスを実行していることになり、処理の内容も安定しないことがわかった。今回ファイルの数を一つ上

げる処理では、読み出しと書き込みでそれぞれ別でファイルを開いていたので、複数のプロセスが、無秩序にアクセスしたためこのようなことが起こったのではないかと考えた。

3 課題 3-2-1

課題 3 4.1 節の pipe プログラムはパイプを一つ用意して、子プロセスから文字列を親プロセスに送信し親プロセスで文字列を出力するというプログラムである。今回 two-way-pipe プログラムでは、パイプを 2 つ用意し、双方向に文字列を送信できるように拡張した。

3.1 設計の説明

一つのパイプを送信、受信の両方に用いることは望ましくないため、パイプの方向を 1 方向に定める。このとき、利用しない方のファイル記述子は close() しておく。このようなパイプを 2 つ用意した。その部分に当たるコードを下に示す。

```
if (pipe(fdCTP) == -1) {
  perror("pipe(Child to Parent) failed.");
  exit(1);
}
if (pipe(fdPTC) == -1) {
  perror("pipe(Parent to Child) failed.");
  exit(1);
}
```

子プロセスから親プロセスへのパイプと、親プロセスから子プロセスへのパイプを作成している。子プロセスにおけるコードを下に示す。

```
if (pid == 0) { /* Child process */
  close(fdPTC[1]);
  close(fdCTP[0]);
  msglenC = strlen(argv[1]) + 1;
  if (write(fdCTP[1], argv[1], msglenC) == -1) {
    perror("pipe write.(Child)");
    exit(1);
  }
  if (read(fdPTC[0], bufC, BUFSIZE) == -1) {
    perror("pipe read.(Child)");
    exit(1);
  }
  printf("Message from parent process: \n");
  printf("\t%s\n",bufC);
  exit(0);
```

まず親プロセスから子プロセスへのパイプの送信のファイル記述子と、子プロセスから親プロセス

へのパイプの受信のファイル記述子を閉じている。プログラム実行時に指定した第 2 引数の文字列を親プロセスへのパイプを用いて送信している。次に、親プロセスからのパイプから文字列を受信して、出力している。親プロセスにおけるコードを下に示す。

```
  } else { /* Parent process */
    close(fdPTC[0]);
    close(fdCTP[1]);
    msglenP = strlen(argv[2]) + 1;
    if (write(fdPTC[1], argv[2], msglenP) == -1) {
      perror("pipe write.(Parent)");
      exit(1);
    }
    if (read(fdCTP[0], bufP, BUFSIZE) == -1) {
      perror("pipe read.(Parent)");
      exit(1);
    }
    printf("Message from child process: \n");
    printf("\t%s\n",bufP);
    wait(&status);
  }
```

まず子プロセスから親プロセスへのパイプの送信のファイル記述子と、親プロセスから子プロセスへのパイプの受信のファイル記述子を閉じている。プログラム実行時に指定した第 3 引数の文字列を子プロセスへのパイプを用いて送信している。次に、子プロセスからのパイプから文字列を受信して、出力している。

3.2 実行結果

```
r-koyama@exp007:~/enshuuC/kadai3$ ./two-way-pipe kodomo oya
Message from child process:
    kodomo
Message from parent process:
    oya
```

第二引数には子プロセスから親プロセスへのメッセージ、第三引数には親プロセスから子プロセスへのメッセージを与えている。正確に送受信が行われていることがわかる。

3.3 考察等

送受信には同じパイプを用いない方がよいということがわかった。write() や read() は送受信を待つので、親プロセスと子プロセスの同期がとりやすい。

4 課題 3-2-2

課題 3 に添付された mergesort.c は再帰を用いて、要素が整数の配列をマージソートで整列するプログラムである。本課題では、 mergesort.c を拡張し、配列を 2 分割しそれぞれの配列に対して並列にソートし、最後に 2 つの配列をマージして、マージソートを完成させるプログラムにした。

4.1 設計の説明

プロセスを生成し、配列全体の前半を子プロセスに、後半を親プロセスにソートさせるようにした。各プロセスでソートを終えた後、子プロセスから親プロセスへ整列済みの全体の前半部分をパイプを用いて送信し、親プロセスで前半部分と後半部分をマージする。実際のコードを下に示す。

```
if (pid == 0) { /* Child process */
  close(fd[0]);
  m_sort(numbers, temp, 0, array_size/2-1);
  if (write(fd[1], numbers, sizeof(int)*(array_size/2)) == -1) {
    perror("pipe write.");
    exit(1);
  }
  exit(0);
} else { /* Parent process */
  close(fd[1]);
  m_sort(numbers, temp, array_size/2, array_size-1);
  if (read(fd[0], numbers, sizeof(int)*(array_size/2)) == -1) {
    perror("pipe read.");
    exit(1);
  }
  merge(numbers, temp, 0, array_size/2, array_size-1);
}
```

まず各プロセスで、安全のために使用しないファイル記述子を閉じている。

4.2 実行結果

```
r-koyama@exp007:~/enshuuC/kadai3$ ./mergesort
Done with sort.
6230623
11732103
11821100
44675796
54527156
56375363
61366340
123718196
124294842
163701158
178821998
227934292
238330100
244602541
257447922
271393840
308437099
中略
1798106229
1811270264
1818885731
1833682104
1841652853
1841953965
1860024296
1896481121
1929257273
1944562882
1946891332
1958712432
2014330568
2044874731
2069888257
2083661007
2092958431
2112030405
```

要素数 100 の配列でプログラムを実行した。ソートが実現されていることがわかる。

4.3 考察等

並列でマージソートを行うようにしたが、ソートにかかる時間は CPU が並列処理ができるかどうかによって異なる。並列処理ができない場合は、並列処理しない場合と計算時間は変わらない。並列処理できる場合でも計算時間は半分にはなるが、オーダーが変わるわけではない。

5 課題 3-3-1

課題 3-3-2 のプログラム alarm.c は、入力した文字列をそのまま返すプログラムであり、 alarm() を用いて、10 秒間何も入力がなければ、タイムアウトとしてプログラムを終了するようなプログラムである。本課題では、 alarm() ではなく、自分で定義した myalarm() を用いて、同様の機能を持つプログラムを作成した。

5.1 設計の説明

myalarm() の始めは、子プロセスの id ををを格納する静的な変数と子プロセスから親プロセスの id を得るための変数を宣言している。次に、子プロセスが生成されているなら、そのプロセスに SIGINT を送り終了させる。次に、 sigaction() を用いて、ゾンビプロセスを生成しない処理を行っている。その後、再び子プロセスを生成し、10 秒待ち、親プロセスに SIGALRM を送る。SIGALRM が送られると “This program is timeout.\n” を出力し、終了するようになっている。関数 myalarm() の箇所のコードを下に示す。

```
  if(pid > 0){
    kill(pid, SIGINT);
  }

  struct sigaction sa;
  sa.sa_handler = SIG_IGN;
  sa.sa_flags = SA_NOCLDWAIT;
  if (sigaction(SIGCHLD, &sa, NULL) == -1) {
    exit(1);
  }

  if ((pid=fork())== -1) {
    perror("fork failed.");
    exit(1);
  }

  if (pid == 0) { /* Child proces */
    p_pid = getppid();
    sleep(sec);
    kill(p_pid, SIGALRM);
    exit(0);
  }
```

5.2 実行結果

```
r-koyama@exp007:~/enshuuC/kadai3$ ./alarm
a
echo: a
b
echo: b
c
echo: c
d
echo: d
This program is timeout.
```

4 つ目の文字列を入力した後に 10 秒間何も入力しなかった。その結果プログラムが終了している。

```
r-koyama@exp007:~/enshuuC/kadai3$ ./alarm
e
echo: e
f
echo: f
g
echo: g
h
echo: h
^Z
[1]+  停止                  ./alarm
r-koyama@exp007:~/enshuuC/kadai3$ ps
  PID TTY          TIME CMD
 1170 pts/4    00:00:00 bash
18387 pts/4    00:00:00 alarm
18525 pts/4    00:00:00 alarm
18605 pts/4    00:00:00 ps
r-koyama@exp007:~/enshuuC/kadai3$ fg
./alarm
This program is timeout.
```

4 つの文字列を 10 秒以内に連続で入力し、 ps コマンドで動いているプロセスを確認した。ゾンビプロセスは生きていないことがわかる。

5.3 考察等

子プロセスをゾンビプロセスとして残さないようにするのに苦労した。また、wait() を行わずに子プロセスの排他的な存在の状態を作るのにも苦労した。

6 課題 3-3-2

課題 3-3-1 の myalarm() を用いて、課題 2 で作成した simple-talk-client プログラムにタイムアウト機能を実装した。

6.1 設計の説明

10 秒間何も入力がなければ、SIGALRM を送る処理は、課題 3-3-1 と同じである。SIGALRM が送られた時の処理は、timeout() であるが、関数 timeout() では、フラッグである変数を立てるにとどまる。フラグが立っていると、ソケットを閉じ、タイムアウトの旨が伝わるように出力し、プログラムを終了する処理を追加した。その部分に当たるコードを下に示す。

```
    if(flag){
      close(sock);
      printf("This program is timeout.\n");
      exit(0);
    }
```

6.2 実行結果

```
r-koyama@exp007:~/enshuuC/kadai3$ ./simple-talk-client exp001
connected
a
exp001 : b
c
exp001 : d
exp001 : e
This program is timeout.
```

サーバー側からの入力があった後 10 秒間何も入力しなかった。その結果プログラムが終了している。

```
r-koyama@exp007:~/enshuuC/kadai3$ ./simple-talk-client exp001
connected
f
g
exp001 : h
i
^Z
[1]+  停止                  ./simple-talk-client exp001
r-koyama@exp007:~/enshuuC/kadai3$ ps
  PID TTY          TIME CMD
 1170 pts/4    00:00:00 bash
21820 pts/4    00:00:00 simple-talk-cli
21968 pts/4    00:00:00 simple-talk-cli
22039 pts/4    00:00:00 ps
r-koyama@exp007:~/enshuuC/kadai3$ fg
./simple-talk-client exp001
This program is timeout.
```

10 秒以内の連続した入力の後、 ps コマンドで動いているプロセスを確認した。ゾンビプロセスは生きていないことがわかる。

6.3　考察等

シグナルハンドラ内では “非同期安全な関数” のみを使用するべきであり、変数の型も制限されることがわかった。