

情報科学演習 C レポート 3

藤田 勇樹

大阪大学 基礎工学部 情報科学科 ソフトウェア科学コース

学籍番号: 09B16068

メールアドレス: u461566g@ecs.osaka-u.ac.jp

担当教員

小島 英春 助教授

内山 彰 助教授

提出日: 2018 年 7 月 2 日

1 課題 3-1

1.1 概要

この課題では、`fork()` した 4 つのプロセスで順にファイル进行处理するつもりが、競合状態に陥ってしまうプログラム `file-counter` を正しく動作するように修正する。

1.2 改造内容

元々のプログラムでは、まず “0” と書き込んだファイル `counter` を作成した後、プロセスを 4 回 `fork` し、それぞれの子プロセスで `counter` ファイルを開いてそれに書いてある数字を読み取り、1 を足して書き込む、という操作を行う。この操作によりプログラム終了時の `counter` ファイルの内容は “4” になることが想定されている。しかし、元々のプログラムのままでは、プロセス間の同期が取れていないために、あるプロセスが `counter` ファイルを読み込んで書き込むまでに、違うプロセスが `counter` ファイルを読み取ってしまうことがある。その結果、プログラム終了時の `counter` ファイルの内容は、実際には 2 や 3 になっている。

これを修正し排他制御を実現するために、セマフォという同期システムを用いる。セマフォには 0 以上の整数を保存し、これに対し `wait` と `signal` という操作が可能である。`wait` をすると、セマフォの値が 1 以上であれば減らし、0 ならば実行待ちキューに `wait` 操作をしたプロセスをプッシュする。一方 `signal` を行うと、実行待ちキューにプロセスがあればプロセスを一つポップしてそのプロセスの実行を再開し、なければセマフォの値を増やす。

`file-counter` 中でのセマフォの使用方法について述べる。各子プロセスの処理内容は以下の通りである。

1. `wait`
2. `counter` ファイルを読み取り 1 加えた値を書き込む (`count1`)
3. `signal`
4. 子プロセス終了

セマフォの初期値は 1 に設定してあるため、子プロセスを 4 つ `fork` した後、最初に実行される子プロセスは `wait` でセマフォの値を 0 にして処理を継続する。その他の子プロセスが `wait` する時にはセマフォの値は 0 になっているから、処理を一時停止し実行待ちキューに格納される。最初の子プロセスの `count1` の実行が終わり `signal` を行うと、実行待ちキュー内の子プロセスが一つ再開される。このとき最初の子プロセスの `count1` はすでに終了しているため、ファイル `counter` の値は 1 になっている。これを繰り返すと、それぞれの子プロセスは前の子プロセスの `count1` の終了を待って自分の `count1` を行うことになるため、`counter` ファイルの最終的な値は 4 になる。

1.3 実行結果

```
$ ./file-counter
count = 1
count = 2
count = 3
```

```
count = 4
$ ./file-counter
count = 1
count = 2
count = 3
count = 4
$ ./file-counter
count = 1
count = 2
count = 3
count = 4
```

プロセス間の同期がとれたことで、確実に count の値が 1 ずつ増え、何度実行しても結果が変わらなくなった。

2 課題 3-2-1

2.1 概要

この課題では、パイプで子プロセスから親プロセスへ文字列を送信するプログラムを拡張し、逆に親プロセスから子プロセスへも同時に送信するプログラム two-way-pipe.c を作成する。

パイプとは、プロセス間の通信に用いるために使用する領域で、ファイルディスクリプタを通じてアクセスするため一種のファイルのように扱うことができる。パイプは `pipe()` システムコールで生成し、`read()` や `write()` を使用して読み書きが可能である。

2.2 改造内容

元々のプログラムでは、子プロセスから親プロセスへのパイプのみ生成されているため、同様にして親プロセスから子プロセスへのパイプを作成する。また、子プロセスではパイプへの `write`、親プロセスではパイプからの `read` を行っているが、それぞれを両方のプロセスで行うようにする。

2.3 実行結果

```
$ ./two-way-pipe hello HELLO
message from child process:
hello
message from parent process:
HELLO
```

3 課題 3-2-2

3.1 概要

この課題では、マージソートのプログラムを拡張し、2 プロセスで並列に手分けしてソートし、それぞれの結果をパイプを用いて一つのプロセスにまとめ、最後にそれらをマージするプログラムを作成する。

3.2 改造方法

まず、マージソートの最初のステップでは配列を 2 分割するが、このうち一つを親プロセスで、もう一つを子プロセスでそれぞれマージソートするようにする。これが終了すると通常のマージソートにおける最後のマージ直前の状態になる。その後、子プロセスでマージソートした後の配列をパイプを経由して親プロセスに送信し、親プロセスが自分でマージソートした配列とパイプで受信した配列をマージすれば、マージソートが完了する。

```
$ ./pipemerge
Done with sort.
52892392
262577817
627763308
700818707
868170278
984698191
1126456746
1162516543
1607248803
1692208745
```

正しくソートできている。

4 課題 3-3-1

4.1 概要

この課題では、システムコール `alarm()` の機能を実現する関数 `myalarm` を作成する。

4.2 実装方法

`myalarm` 内で `fork()` を行い、親プロセスはそのまま呼び出し元に戻り、子プロセスは指定された秒数だけ `sleep()` した後親プロセスに `SIGALRM` を送ればよい。

ただし、単純にこれを行うだけではいくつかの問題がある。まず、要求仕様上、`myalarm` が `SIGALRM` を送る前に `myalarm` を再度呼び出すと `myalarm` のタイマーをリセットしなければならない。これを実現するた

めに、前回の myalarm 呼び出し時に作成した子プロセスのプロセス ID を static 変数で記憶しておき、再度呼び出された際は前回の子プロセスに SIGKILL を送り中断させる。

もう一つの問題として、通常 fork() で作成した子プロセスが使用するリソースは子プロセスの終了後親プロセスが wait() することで解放されるが、myalarm を呼び出して戻ってきた親プロセスは続きの処理を行うため、子プロセスを wait() することができない。そのため子プロセスは SIGALRM を送った後ゾンビ状態になってしまう。これを解決するために、sigaction() システムコールを用いる。これは signal() システムコールのようにシグナルへの応答を設定するものであるが、より詳細な動作を設定することができる。ここでは、シグナルハンドラとしてそのシグナルを無視する (何もしない) SIG_IGN を設定し、さらに、追加の動作として、設定したシグナルが SIGCHLD の場合、子プロセスが終了したときにゾンビ状態にしないようにする SA_NOCLDWAIT を設定する。これにより親プロセスが wait() しなくても子プロセスは終了後ゾンビ状態にならない。

4.3 実行結果

```
$ ./alarm
(10 秒経過)
This program is timeout.
(プログラム終了)
$ ./alarm
a
echo: a
(a を入力してから 10 秒経過)
This program is timeout.
(プログラム終了)
$
```

アラームが正しく動作し、アラームのリセットもできている。また、ps コマンドで確認するとタイムアウト後子プロセスもゾンビプロセスにならずに終了している。

5 課題 3-3-2

5.1 概要

この課題では、前節で作成した myalarm を、前回の課題で作成した simple-talk-client に組み込み、一定時間発言しなかった場合にタイムアウトする機能を作成する。

5.2 改造内容

simple-talk-client では標準入力とソケットを同時に監視するため select() システムコールを用いているが、これの実行中にシグナルハンドラの割り込みが発生すると、select() はエラー扱いとなり、-1 を返して errno を EINTR に設定する。よってこの状態になった時、ソケットを close() してプログラムを終了すればよい。

5.3 実行結果

前回の拡張課題として実装した，発言者の名前を表示する機能はそのままにしてある．

```
$ ./simple-talk-client exp101
Enter your name>CLT
connected with SVR
CLT > hello
(hello 入力後 10 秒経過)
connection timed out.
closed
(プログラム終了)
$
```

6 発展課題 1

6.1 概要

この課題では，セマフォを利用して，すべてのプロセスが処理中のある時点までたどり着くまで他のプロセスは待機するバリア同期を実装する．

6.2 仕様

このプログラムはまず，引数で与えられた数だけ `fork()` し子プロセスを作成する．各子プロセスは，自身のプロセス ID を 5 で割った余りの秒数だけ `sleep()` し，その後 “Child process (pid) ended” を出力し終了する．親プロセスはすべての子プロセスが終了したのを確認してから，“All child processes ended” を出力して終了する．

6.3 実装方法

各子プロセスは終了時に `signal` を行い，親プロセスは `signal` を子プロセスの数だけ `wait` するようにする．これにより，親プロセスはすべての子プロセスの終了を待つことができる．これは `wait` 用の `sembuf` の `sem_op` を -1 から -n (n は子プロセスの数) にすれば実現できる．

6.4 実行結果

```
$ ./barrier 10
Child process 14975 ended
Child process 14980 ended
Child process 14976 ended
Child process 14971 ended
```

```
Child process 14977 ended
Child process 14972 ended
Child process 14978 ended
Child process 14973 ended
Child process 14979 ended
Child process 14974 ended
All child processes ended
```

すべての子プロセスの終了の出力の後に親プロセスが出力を行っている。

7 発展課題 3

7.1 概要

この課題では、ここまでの課題で学んだセマフォやシグナルを用いて、2 プロセスが交互に動作するようなプログラムを作成する。

7.2 仕様

プログラムを起動すると、`fork()` したプロセス A と B が、あらかじめランダムに作成した配列 A と B の内容を交互に表示する。プロセス A は配列 A を、プロセス B は配列 B を表示する。

7.3 実装方法

セマフォが 1 つだと、次にどちらのプロセスが実行されるか分からないため、セマフォを 2 つ用いた。具体的には、プロセス A では、

1. セマフォ 1 を wait
2. A の内容を一つ出力
3. セマフォ 2 に signal

の処理を繰り返し、プロセス B では、

1. セマフォ 2 を wait
2. B の内容を一つ出力
3. セマフォ 1 に signal

を繰り返すようにした。ここで、セマフォ 1 の初期値を 1 に、セマフォ 2 の初期値を 0 にすると、

1. プロセス A が wait から復帰、処理を行う (セマフォ 1=0, セマフォ 2=0)
2. プロセス A が signal(セマフォ 1=0, セマフォ 2=1)
3. プロセス B が wait から復帰、処理を行う (セマフォ 1=0, セマフォ 2=0)
4. プロセス B が signal(セマフォ 1=1, セマフォ 2=0)

これを繰り返すことになり、仕様通りの動作が行える。

7.4 実行結果

```
$ ./rotation
A[0] = 2899
B[0] = 6064
A[1] = 4806
B[1] = 5391
A[2] = 8125
B[2] = 9354
A[3] = 3773
B[3] = 6435
A[4] = 8643
B[4] = 1452
(中略)
B[96] = 2722
A[97] = 5798
B[97] = 8942
A[98] = 5284
B[98] = 6991
A[99] = 1253
B[99] = 991
```

正しく交互に出力できている。

8 感想

今回扱ったパイプや前回扱ったソケット等はファイルディスクリプタを通じて一種のファイルとして扱えるため、統一感があり非常に使いやすいものだと感じた。