

情報科学演習C 課題3レポート

氏名 山久保孝亮
所属 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
メールアドレス u327468b@ecs.osaka-u.ac.jp
学籍番号 09B22084
提出日 2024年6月30日
担当教員 平井健士, 中島悠太

1 課題 3-1

1.1 アルゴリズム

この課題の処理の流れは以下図 1 のフローチャートのとおりである。

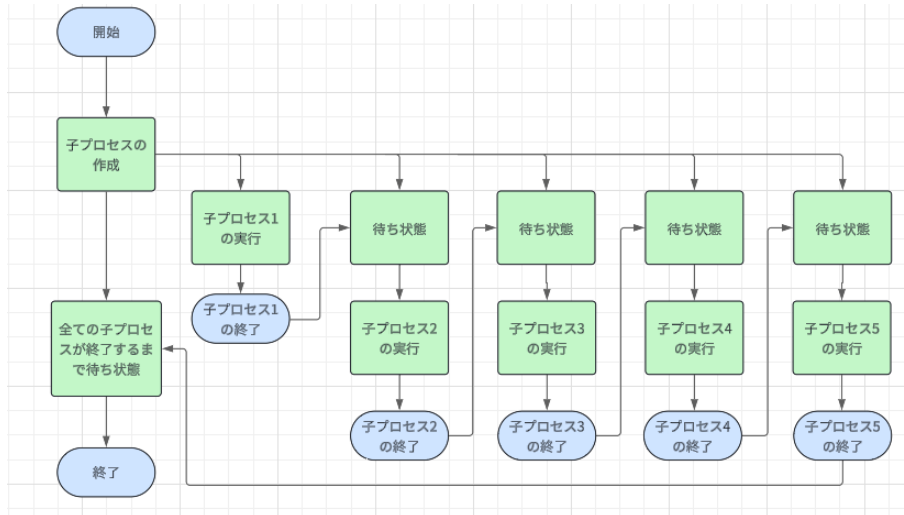


図 1: 課題 3-1 の処理の流れ

今回の file-counter プログラムでは子プロセスを 4 つ作成し、それぞれの子プロセス内の処理を順番に一つずつ実行していく必要がある。そのため、実行している子プロセス以外は待ち状態にしておき、実行中のプロセスが終了してから一つだけ待ち状態を開放して処理を実行するというアルゴリズムで今回の課題を実装した。フローチャートにおいて、子プロセスの終了から待ち状態へ伸びている矢印は子プロセスが終わってから矢印の先の待ち状態が解放されるということを表している。今回の課題のクリティカルセクションは後述の count1 関数なので、この直前に待ち状態と解放を処理する関数を呼び出すようにした。

1.2 実装方法

ここではセマフォの処理について中心に記述する。以下は親プロセスのプログラムを抜粋したものである。

```
1 if ((key = ftok(".", 1)) == -1){
2     fprintf(stderr, "ftok path does not exist.\n");
3     exit(1);
4 }セマフォセットを作成
5 //
6 if ((sem_id=semget(key, 4, 0666 | IPC_CREAT)) == -1) {
7     perror("semget error.");
8     exit(1);
9 }
10 // セマフォの初期値を1に設定
11 if (semctl(sem_id, 0, SETVAL, 1) == -1) {
12     perror("semctl SETVAL error.");
13     exit(1);
14 }
15 count = 0;
16
```

```

17 if ((ct=fopen(filename, "w"))==NULL) exit(1);
18 fprintf(ct, "%d\n", count);
19 fclose(ct);
20 for (i=0; i<NUMPROCS; i++) {
21     if ((pid=fork())== -1) {
22         perror("fork failed.");
23         exit(1);
24     }
25     if (pid == 0) { 子プロセスの処理のため省略
26     }
27 }
28 for (i=0; i<NUMPROCS; i++) {
29 wait(&status);
30 }
31
32 // セマフォを削除
33 if (semctl(sem_id, 0, IPC_RMID) == -1) {
34     perror("semctl IPC_RMID error.");
35     exit(1);
36 }
37 exit(0);

```

即ち、今回のプログラムにおける親プロセスの処理の流れは以下になる。

1. セマフォセットの初期設定
2. 子プロセスの作成
3. 子プロセスがすべて終了するまで待つ
4. セマフォの削除

以下でその詳細について記述する。

1. まず, `ftok()` を使って `key` を作成する. このとき, 第一引数の `"."` は現在のディレクトリを表し, 第二引数の `"1"` はプロジェクトを一意に識別する文字を指定している.[1] 次に `semget()` を使ってセマフォセットを作成している. 第一引数は作成した `key`, 第二引数はセマフォの数である 4, 第三引数はアクセス許可の定義として使用され, 全てに読み込み, 書き込み許可を与えるという設定である.[2] 最後に `semctl()` を使って 0 番目のセマフォの値を 1 に設定する. 第一引数はセマフォIDの `sem_id` を, 第二引数でセマフォ番号の 0 を, 第三引数の `SETVAL` は `semval` の値を第四引数で指定された値に設定する制御操作を指定するパラメータ値である. セマフォの初期値を 1 に設定する理由については子プロセスの処理にて詳細を記述する.
2. `fork()` を使って子プロセスを作成する.`for` 文の中で呼び出すことによって子プロセスを複数作成する.`fork()` の戻り値を `pid` に格納し,`pid` の値によって子プロセスを分岐させて親プロセスと子プロセスの処理内容を分ける. ここでは親プロセスについての記述なので, 以下の 3,4 の記述は親プロセスの処理についてである.
3. `wait()` を使って子プロセスの終了を待つ. 引数には状況の情報値を格納する.2 で作成したプロセスの数だけ `for` 文でこれを繰り返すことですべてのプロセスが終了するまで待ち続けることができるようになる.[3]

- 最後にセマフォを `semctl()` を使って削除する. 第三引数の `IPC_RMID` は第一引数によって指定されたセマフォID をシステムから除去し, それに関連するセマフォセットを破棄する.[2]

また, 子プロセスのプログラムは以下のようになる.

```
1 if (pid == 0) { /* Child process */
2     lock(sem_id);
3     count = count1();
4     printf("count = %d\n", count);
5     unlock(sem_id);
6     exit(0);
7 }
```

即ち, 以下のような流れで処理が実行される.

1. `lock()` の呼び出し
2. `count1()` の呼び出し
3. `unlock()` の呼び出し
4. 子プロセスの終了

以下でその詳細について記述する.

1. `pid` の値が 0 のときは子プロセスであると判定できる. 子プロセス内の処理がクリティカルセクションであるので, まず `lock()` を呼び出す. `lock()` のプログラムは以下のようになっている.

```
1 void lock(int semid){
2     struct sembuf op;
3     op.sem_num = 0;
4     op.sem_op = -1;
5     op.sem_flg = 0;
6     if (semop(semid, &op, 1) == -1) {
7         perror("semop lock");
8         exit(1);
9     }
10 }
```

ここでは `semop()` を使ってプロセスを停止している. 第一引数はセマフォID の `semid`, 第二引数は `sembuf` 構造体のポインタ, 第三引数は第二引数の数を表す. `sembuf` 構造体は 3 津のメンバが存在し, `sem_num` はセマフォ番号, `sem_op` はセマフォ操作, `sem_flg` は操作フラグを表す.[4] `sem_op` は -1, それ以外は 0 に指定することによってセマフォ値が `sem_op` の値を足した結果 0 以上にならない場合はプロセスを停止する. セマフォ値の初期値を 1 にしていたことによって, 最初のプロセスは停止しないが, 次のプロセスはセマフォ値が 0 で `sem_op` を加算すると負の値になってしまうので停止する.

2. セマフォに関する処理はないので省略する.
3. `unlock()` のプログラムは以下の通りである.

```
1 void unlock(int semid){
2     struct sembuf op;
3     op.sem_num = 0; // セマフォの番号を固定値として指定
4     op.sem_op = 1; // 操作 (加算、アンロック)
5     op.sem_flg = 0;
```

```

6     if (semop(semid, &op, 1) == -1) {
7         perror("semop V");
8         exit(1);
9     }
10 }

```

まず `semop()` を使ってセマフォ値を変更し、子プロセスの待ち状態を一つ開放する。第二引数の `sembuf` 構造体の `sem_op` を 1 に、それ以外を 0 に設定することによって待っている一つの状態の子プロセスにおいて -1 を加算してもセマフォ値が 0 以上になるので待ち状態ではなくなる。このようにして待ち状態のプロセスの開放を一つずつ行うことによってクリティカルセクションに複数のプロセスが同時にアクセスできないようにしている。

4. `exit()` を使って子プロセスを終了する。

1.3 実行結果

この実行結果は以下ようになる。以上の結果により、きちんと `count` が 1 ずつ増やされ、最終的に

```

k-yamakb@exp017:~/enshuC/kadai3$ ./file-counter
count = 1
count = 2
count = 3
count = 4

```

図 2: ターミナル上の実行結果

```

counter
1 4

```

図 3: counter ファイルの内容

counter ファイルには 4 が記入されているということがわかる。

2 課題 3-2-1

2.1 アルゴリズム

この課題の処理の流れは以下図 4 のフローチャートのとおりである。

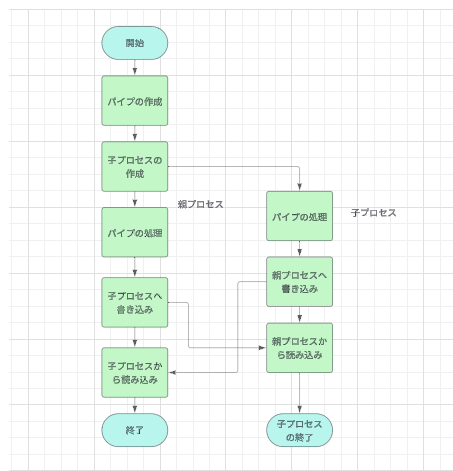


図 4: 課題 3-2 の処理の流れ

今回の two-way-pipe プログラムでは子プロセスを一つ作成し、子プロセスから親プロセスへ最初の引数の

文字列を、親プロセスから子プロセスへ次の引数の文字列を送信する。そして親プロセスは子プロセスから、子プロセスは親プロセスから送られてきた文字列を受け取り表示させる。図の左側は親プロセス、右側は子プロセスの処理を表す。双方向のパイプを実現するために、子プロセス用のパイプと親プロセス用のパイプの二つのパイプを作成して実装した。

2.2 実装方法

パイプ部分を中心に実装方法を記述する。記述する内容としては以下のとおりである。

1. パイプの作成
2. 子プロセスの処理
3. 親プロセスの処理

以下でその詳細について記述する。

1. パイプの作成は、以下のようなプログラムになる。

```
1 if (pipe(parenttochild) == -1 || pipe(childtoparent) == -1) {
2     perror("pipe failed.");
3     exit(1);
4 }
```

pipe() を使い、親プロセスから子プロセスへ送信したり、親プロセスが読み込むためのパイプとして parenttochild、子プロセスから親プロセスへ送信したり、子プロセスが読み込むためのパイプとして childtoparent を定義した。パイプを二つ作成した理由としては、パイプは安全のために書き込み側か読み込み側のどちらかをクローズする必要があるため、一つのパイプではどちらの機能も同時に使うことができないためである。

2. fork() を使って子プロセスを一つ作成し、pid が 0 かどうかで親プロセスと子プロセスの処理を分離した。子プロセスのプログラムは以下の通りである。

```
1     if (pid == 0) { /* Child process */
2         close(parenttochild[1]);
3         close(childtoparent[0]);
4         msglen = strlen(argv[1]) + 1;
5         if (write(childtoparent[1], argv[1], msglen) == -1) {
6             perror("pipe write.");
7             exit(1);
8         }
9         if (read(parenttochild[0], buf, BUFSIZE) == -1) {
10            perror("pipe read.");
11            exit(1);
12        }
13        printf("Message from parent process: %s\n", buf);
14        wait(&status);
15        exit(0);
16    }
```

子プロセスでは parenttochild の書き込み機能と childtoparent の読み込み機能を使用しないので、まずこれらを close() しておく。そして第二引数の文字列を write() を使って送信し read() を使って親プロセスから送られてきた第一引数の文字列を受信して標準出力に出力する。

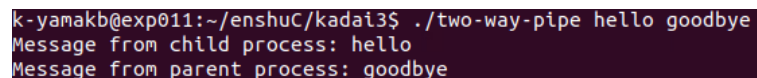
3. 親プロセスのプログラムは以下のとおりである.

```
1     else { /* Parent process */
2         close(parenttochild[0]);
3         close(childtoparent[1]);
4         msglen = strlen(argv[2]) + 1;
5         if (write(parenttochild[1], argv[2], msglen) == -1) {
6             perror("pipe write.");
7             exit(1);
8         }
9         if (read(childtoparent[0], buf, BUFSIZE) == -1) {
10            perror("pipe read.");
11            exit(1);
12        }
13        printf("Message from child process: %s\n", buf);
14        wait(&status);
15    }
```

親プロセスでは parenttochild の読み込み機能と childtoparent の書き込み機能を使用しないのでこれらを close() しておく. そして第一引数の文字列を write() を使って送信し read() を使って子プロセスから送られてきた第二引数の文字列を受信して標準出力に出力する.

2.3 実行結果

以下は two-way-pipe プログラムを実行したときの結果である. これにより, 第一引数で与えられた hello



```
k-yamakb@exp011:~/enshuC/kadai3$ ./two-way-pipe hello goodbye
Message from child process: hello
Message from parent process: goodbye
```

図 5: two-way-pipe.c の実行結果

が子プロセスから, 第二引数で与えられた goodbye が親プロセスから送られてきていることがわかる.

3 課題 3-2-2

3.1 アルゴリズム

この課題の処理の流れは以下図 6 のフローチャートのとおりである.

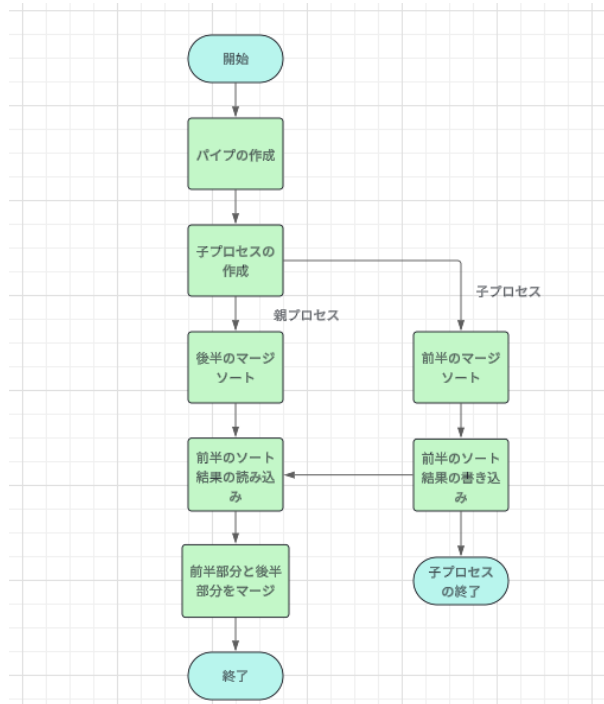


図 6: 課題 3-2 の処理の流れ

今回の mergesort プログラムでは子プロセスを一つ作成し、子プロセスでソート対象の配列の前半半分を、親プロセスでソート対象の配列の後半半部分をソートしてからマージするというアルゴリズムで作成した。図 6 の左側が親プロセス、右側が子プロセスに対応する。また、子プロセスのソート結果を親プロセスに伝えるためにパイプを使用した。これにより前半と後半のソートを並列に実行できるようになる。

3.2 実装方法

ここではマージソートのアルゴリズムについては言及せず、どのようにして並列化をしたかについて中心に記述する。今回、変更を加えたのは mergesort() 内のみである。関数 mergesort 中に図 5 のフローチャートの流れで並列化を実装したのでそれぞれの詳細について記述する。

1. パイプの作成では、pipe() を使ってパイプ fd を作成した。
2. プロセスの作成では、fork() を使って子プロセスを作成し、pid が 0 かどうかで親プロセスと子プロセスの処理を分離した。子プロセスのプログラムは以下ようになる。

```

1  if(pid == 0){/*child process*/
2      close(fd[0]);
3      m_sort(numbers, temp, 0, (array_size-1)/2);
4      if (write(fd[1], numbers, ((array_size - 1) / 2 + 1) * sizeof(int))
        ==-1) {
5          perror("pipe write.");
6          exit(1);
7      }
8      exit(0);
9  }

```

子プロセスでは fd の読み込み機能は使用しないので、まず close() しておく。そして引数には 0 と配列の要素の中央値を渡して msort() 呼び出す。これによってソート対象の配列の前半部分をマージソートすることができる。次に write() を呼び出して前半がソートされた配列を親プロセスに送信する。このときに引数に 0 と配列の要素の中央値を渡すことで配列全体ではなく前半部分のみを送ることができる。配列全体を送ってしまうと親プロセスでソートされた部分と競合してしまい、正しいソート結果ではなくなってしまう。

3. 親プロセスのプログラムは以下のようになる。

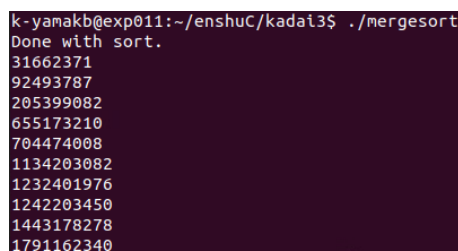
```
1     else{/*parent pocess*/
2         close(fd[1]);
3         m_sort(numbers, temp, (array_size-1)/2+1, array_size - 1);
4         if (read(fd[0], numbers, ((array_size - 1) / 2 + 1) * sizeof(int)) ==-1)
5             {
6                 perror("pipe read.");
7                 exit(1);
8             }
9         merge(numbers, temp, 0, (array_size-1)/2+1, array_size-1);
10    }
```

親プロセスでは fd の書き込み機能は使用しないので close() しておく。親プロセスでは引数として配列の要素数の中央値と配列の要素数を渡して msort() を呼び出す。これによってソート対象の配列の後半部分をマージソートすることができる。その後 read() を使って子プロセスから送信された前半のソート結果を読み込む。このときに引数に配列の 0 と配列の要素数の中央値を渡すことで前半部分だけを読み込む。

4. 最後に前半部分と後半部分を mergesort() を呼び出してソートする。

3.3 実行結果

以下は mergesort を実行したときの結果である。このことから、昇順に正しくソートすることができてい



```
k-yamakb@exp011:~/enshuC/kadai3$ ./mergesort
Done with sort.
31662371
92493787
205399082
655173210
704474008
1134203082
1232401976
1242203450
1443178278
1791162340
```

図 7: mergesort.c の実行結果

ると考えられる。

4 課題 3-3-1

4.1 アルゴリズム

この課題の処理の流れは以下図 8 のフローチャートの通りである。
今回の課題では alarm プログラム内の myalarm() を alarm() を使用せずに実現するというものである。

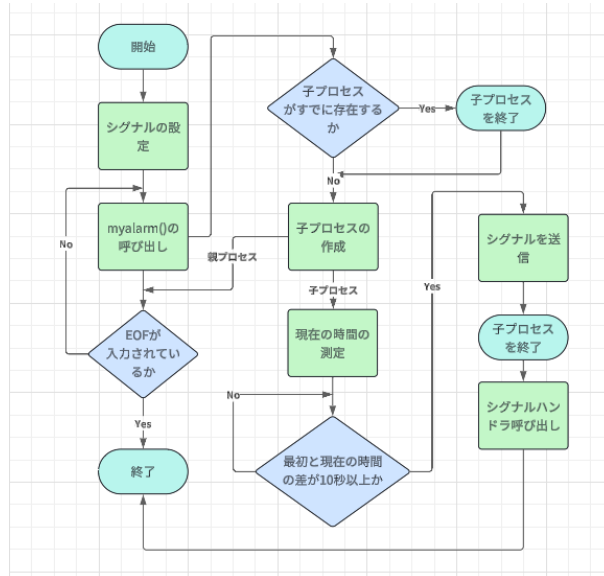


図 8: 課題 3-3-1 の処理の流れ

alarm() は子プロセスが引数で指定した秒数 sec だけ経つとシグナルを送信し, 親プロセス内でシグナルハンドラが呼び出される. 上図フローチャートの左部が main() の処理, 中央と右部が alarm() の処理を表している.

4.2 実装方法

今回の課題では myalarm() 内の実装のみを求められているので, myalarm() の実装方法について詳細に記述する. alarm() のプログラムは以下ようになる.

```

1 void myalarm(int sec) {
2     if (timer_pid > 0) {
3         kill(timer_pid, SIGTERM); // 子プロセスを終了させる
4         waitpid(timer_pid, NULL, 0); // 子プロセスの終了を待つ (ゾンビプロセス防止)
5     }
6     int pid;
7     time_t start_time, current_time;
8     int elapsed_seconds = 0;
9     if ((pid=fork())== -1) {
10         perror("fork failed.");
11         exit(1);
12     }
13     if(pid==0)子プロセス{/**/
14         time(&start_time);
15         while (elapsed_seconds < sec) {
16             time(&current_time);
17             elapsed_seconds = difftime(current_time, start_time);
18             sleep(1);
19         }
20         if (kill(getppid(),SIGALRM) == -1) {
21             perror("kill failed.");
22             exit(1);

```

```

23     }
24     exit(0);
25 }else{
26     timer_pid = pid;
27 }
28 }

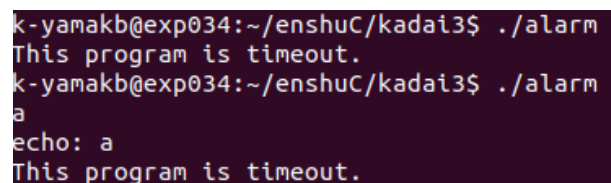
```

以下ではその処理を順番に記述する.

1. フローチャートから, myalarm() では最初にすでに子プロセスが存在しているかどうかを判定する. 上のプログラムの 2 から 5 行目にこの処理を記述している. fork() の戻り値をタイマー管理用のプロセス ID を格納しているグローバル変数 timer_pid に格納している. この変数は最初に -1 に初期化されているので, myalarm() の呼び出しが初めてのかどうかを子の変数の値によって判定することができる. まず最初に timer_pid が正であるかどうかを判定している. fork() の戻り値が格納されているので, 正であれば子プロセスが存在する, つまり初めての myalarm() の呼び出しではないことがわかる. よって前回作成した子プロセスを終了させ, waitpid() を使って子プロセスの終了を待つことでゾンビプロセスが発生することを避ける.
2. 次に子プロセスを作成する. 時間経過を測定するために fork() を使用し, pid によって条件分岐することで親プロセスと子プロセスの処理を分ける. 25 から 27 行目のように, 親プロセスでは 1 で使用した timer_pid に fork() の戻り値を格納する.
3. 13 から 25 行目のように, 子プロセスでは while 文の繰り返しを一定時間が経過するまで繰り返し続けることで一定時間の計測を実現している. 具体的には, myalarm() を呼び出したときの時刻と while 文のそれぞれの繰り返し時の時刻との差を格納している elapsed_seconds と指定した sec を比較し, 前者の方が大きくなった際にループを抜けるという処理としている. 自刻の計測には time() を使用し, 差の計算には difftime() を使用した. そしてループを抜けた後に kill() を使ってシグナルハンドラを起動するシグナルを送り, 子プロセスを終了する.

4.3 実行結果

以下は alarm プログラムを実行したときの結果である. 一回目の実行では何も入力せずにしばらく待つ



```

k-yamakb@exp034:~/enshuC/kadai3$ ./alarm
This program is timeout.
k-yamakb@exp034:~/enshuC/kadai3$ ./alarm
a
echo: a
This program is timeout.

```

図 9: alarm の実行結果

た. このとき, タイムアウトの文字列が表示されてプログラムが終了した. また, 二回目の実行ではプログラムを実行してから 5 秒たってから文字列 a を入力してからしばらく待った. このとき, a を入力してから 10 秒後にプログラムが終了した. 以上のことから, プログラムが仕様通りに動作していることがわかる.

5 課題 3-3-2

5.1 アルゴリズム

この課題の処理の流れは以下のフローチャートの通りである.

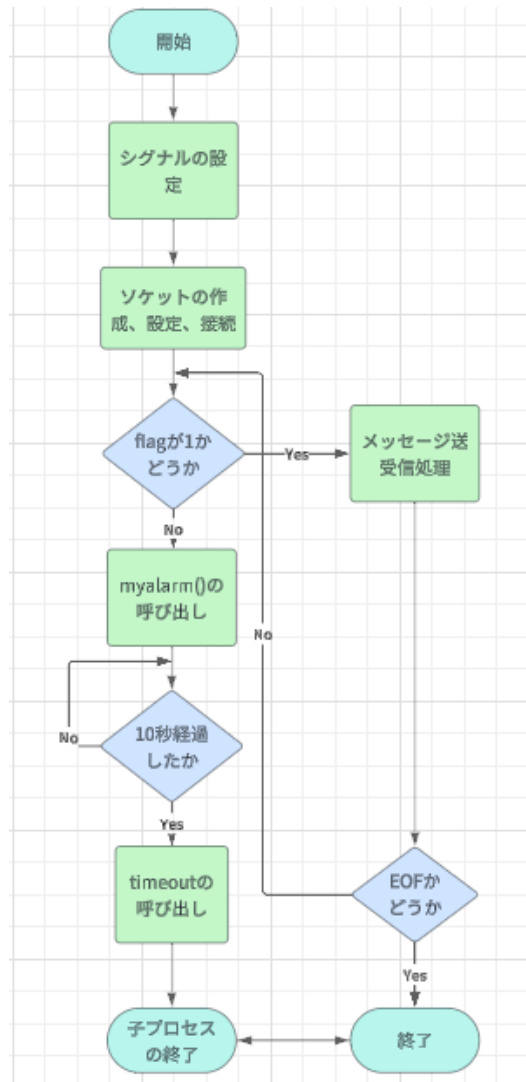


図 10: 課題 3-3-2 の処理の流れ

この課題は大部分が課題 2 の内容なので、以下では追加した点について中心に記述する。図 5 の `myalarm()` の呼び出しの下側は子プロセスの処理、右側は親プロセスの処理となっている。また、それぞれのプロセスの終了から出ている両方向への有向辺は一方のプロセスが終了するともう一方のプロセスの処理も終了するというを示している。これによって、10 秒間経過するか EOF が入力されたときにどちらのプロセスも終了させることができる。

5.2 実装方法

以下では 4 で述べた `myalarm()` を使用するので、この関数については詳細を記述しない。上のフローチャートより、このプログラムが処理する内容は以下のとおりである。

1. シグナル、ソケットの設定
2. 繰り返し文の処理
3. シグナルハンドラ `timeout` の設定

4. タイムアウトの条件分岐

以下でその詳細について述べる。

1. シグナルとソケットの設定は課題 3-3-1, 課題 2 と同様に, 以下のプログラムのように設定した。

```
1  if(signal(SIGALRM,timeout) == SIG_ERR) {
2      perror("signal failed.");
3      close(sock);
4      exit(1);
5  }
6  if ((sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
7      perror("socket");
8      exit(1);
9  }
10
11 /* ホストが存在するか確認*/
12 server = gethostbyname(argv[1]);
13 if (server == NULL) {
14     fprintf(stderr, "ERROR, no such host as %s\n", argv[1]);
15     exit(0);
16 }
17
18 /* ソケットアドレス再利用の指定*/
19 reuse = 1;
20 if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse)) < 0) {
21     perror("setsockopt");
22     exit(1);
23 }サーバ受付用アドレスの設定
24 /**/
25 bzero((char *)&svr, sizeof(svr));
26 svr.sin_family = AF_INET;
27 bcopy((char *)server->h_addr, (char *)&svr.sin_addr.s_addr, server->h_length);
28 svr.sin_port = htons(10130);
29
30 /* ソケットをサーバに接続*/
31 if (connect(sock, (struct sockaddr *)&svr, sizeof(svr)) < 0) {
32     perror("client: connect");
33     close(sock);
34     exit(1);
35 }
36 printf("connected\n");
```

2. 繰り返し文のプログラムは以下ようになる。

```
1  do {
2      myalarm(TIMEOUT);
3      /* 入力を監視するファイル記述子の集合を変数にセットする rfd */
4      FD_ZERO(&rfd);
5      FD_SET(0, &rfd); /* 標準入力*/
6      FD_SET(sock, &rfd); /* ソケット*/
7      /* 監視する待ち時間を秒に設定 10 */
8      tv.tv_sec = 15;
9      tv.tv_usec = 0;
```

```

10      /* 標準入力とソケットからの受信を同時に監視する*/
11      if (select(sock + 1, &rfd, NULL, NULL, &tv) > 0) {
12          if (FD_ISSET(0, &rfd)) {
13              bzero(rbuf, 1024);
14              if (fgets(rbuf, 1024, stdin) == NULL) {
15                  if (timer_pid > 0) {
16                      kill(timer_pid, SIGTERM); // 子プロセスを終了させる
17                      waitpid(timer_pid, NULL, 0); // 子プロセスの終了を待つ (ゾ
18                          ンビプロセス防止)
19                  }
20                  printf("\nEOF detected\n");
21                  break;
22              }
23              n = write(sock, rbuf, strlen(rbuf));
24              if (n < 0) {
25                  perror("ERROR writing");
26                  break;
27              }
28          }
29          if (FD_ISSET(sock, &rfd)) {
30              bzero(rbuf, 1024);
31              n = read(sock, rbuf, 1024);
32              if(n <= 0){
33                  perror("Connection closed by client.\n");
34                  close(sock);
35                  return 0;
36              }else{
37                  printf("%s",rbuf);
38              }
39          }
40      } while (!flag);

```

ここでは大きく分けて以下のような処理が繰り返し実行されている。

- (a) myalarm() の実行
- (b) select() による待ち状態

以下でこれらについて詳細に記述する。

- (a) select() による待ち状態は基本的には前回の課題の内容なので記述しないが、EOF を入力した際の処理には変更を加えたのでそれについて記述する。EOF を入力すると親プロセスが繰り返し文から break し、プログラムが終了するが alarm() を呼び出した後だと親プロセスが終了してから子プロセスが終了するので演習室環境が落ちてしまうことがあった。したがって、繰り返し文から出る前の 15 から 18 行目に課題 3-3-1 のゾンビプロセスをなくす処理と同じ以下のプログラムをつけることでこの問題を解決した。
- (b) alarm() は一度呼び出されてから 10 秒以内にもう一度呼び出すと前回呼び出した際のプロセスを終了する。これによって、何も入力されない状態が 10 秒続けばシグナルが送信されプログラムが終了するという機能が達成される。また、この繰り返し文の繰り返し条件は初期値が 0 のグローバル変数 flag が 1 でないときである。この変数の処理についてはシグナルハンドラにおいて詳細に記述する。

3. シグナルハンドラ `timeout()` のプログラムは以下になる。

```
1 void timeout(){
2     flag = 1;
3 }
```

シグナルハンドラは仕様により非同期安全な関数のみしか使用できない。したがって、`printf` のような関数は使用できないため、今回のプログラムではシグナルハンドラ `timeout` 内では `flag` という変数の値を変更するだけとした。この `flag` は子プロセス内で 10 秒が経過したときに 1 となり、それ以外は 0 のグローバル変数である。これによって先ほどの繰り返し文の条件が変更されるので 10 秒たった時点で親プロセスは繰り返しから抜けることができるようになる。

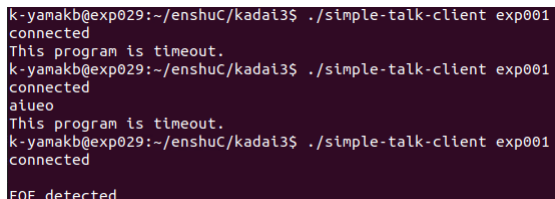
4. タイムアウトが発生すると先ほどシグナルハンドラで立てた `flag` が 1 となる。それによって繰り返し文から抜けた後以下のプログラムが実行される。

```
1 if(flag == 1){
2     printf("This program is timeout.\n");
3 }
4 close(sock);
5 return 0;
```

これによってタイムアウトしてプログラムが終了したことが表示される。

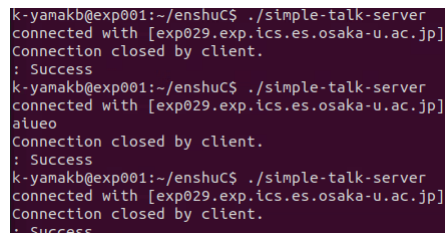
5.3 実行結果

以下は `exp029` というホスト名で `simple-talk-client` を、`exp001` というホスト名で `simple-talk-server` を実行したときの結果である。上の結果では、以下の三通りの方法でプログラムを実行している。



```
k-yamakb@exp029:~/enshuC/kadai3$ ./simple-talk-client exp001
connected
This program is timeout.
k-yamakb@exp029:~/enshuC/kadai3$ ./simple-talk-client exp001
connected
aiueo
This program is timeout.
k-yamakb@exp029:~/enshuC/kadai3$ ./simple-talk-client exp001
connected
EOF detected
```

図 11: ターミナル上の実行結果



```
k-yamakb@exp001:~/enshuC$ ./simple-talk-server
connected with [exp029.exp.ics.es.osaka-u.ac.jp]
Connection closed by client.
: Success
k-yamakb@exp001:~/enshuC$ ./simple-talk-server
connected with [exp029.exp.ics.es.osaka-u.ac.jp]
aiueo
Connection closed by client.
: Success
k-yamakb@exp001:~/enshuC$ ./simple-talk-server
connected with [exp029.exp.ics.es.osaka-u.ac.jp]
Connection closed by client.
: Success
```

図 12: counter ファイルの内容

1. 何も入力せずに待ち続ける。
2. 約 5 秒たってから文字列 `aiueo` を送信して待ち続ける。
3. EOF を入力する。

以下でその実行結果について詳細を述べる。

1. 何もせずに待ち続けると `myalarm()` で作成された子プロセスによってシグナルが送信され、シグナルハンドラが呼び出されてフラグの情報が変更される。これによってプログラムが終了するとともにタイムアウトしたという文字列が表示されている。
2. `aiueo` という文字列を入力しても前回の課題のようにメッセージを送受信することができていることがわかる。また、メッセージを送信してから 10 秒後にプログラムが終了したことから、使用を満たしていることがわかる。

3. EOF を入力しても正常にプログラムが終了していることがわかる.

6 発展課題 1

6.1 アルゴリズム

6.2 実装方法

6.3 実行結果

7 発展課題 2

7.1 アルゴリズム

今回の課題では課題 3-2-2 における, 子プロセスによるマージソート高速化プログラムを改良し, 指定された任意の k 個のプロセスに配列を分解してソートするプログラムを作成した. この課題では `mergesort()` の部分のみを変更したのでその部分のフローチャートを以下に示す. フローチャート中の p は, そのプロセス

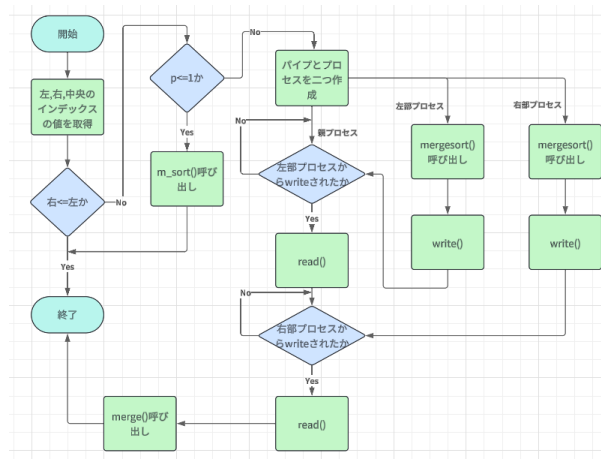


図 13: mergesort() のフローチャート

において作成しなければならないプロセスの個数を表す.

このプログラムでは任意の k 個のプロセスに分解する方法として二分木の形状で再帰的に `mergesort()` を呼び出すという方法を採用した. 具体的には, 以下の手順に従っている.

1. 配列を右部と左部に分け, それぞれに対応する子プロセスを作成する. 以下ではこれらをそれぞれ左部のプロセス 1.1 と右部のプロセス 1.2 と呼ぶ. 各プロセスの数字は二分木におけるの深さと左から何番目であるかに対応するものとする. このとき, 子プロセスを作成して `mergesort()` を呼び出すがその際の p は左部のプロセスでは $p_{left} = \frac{p}{2}$, 右部のプロセスでは $p_{right} = p - p_{left}$ として引数にする.
2. さらにプロセス 1.1 を左部のプロセス 2.1 と右部のプロセス 2.2 に分ける. 右プロセスも同様に分ける.
3. 合計の子プロセスの個数が指定した k に達するまで行う. k に達したかどうかは引数として与えている p の値が子プロセスになるにつれて減少していることから判断する.

実際に 9 個のプロセスに分けると考えたときの二分木の様子は以下ようになる. 黒い葉は $p=1$ のときの子プロセスを表しており, それぞれの白い節点で葉の処理が終わってから `merge()` が呼び出される.

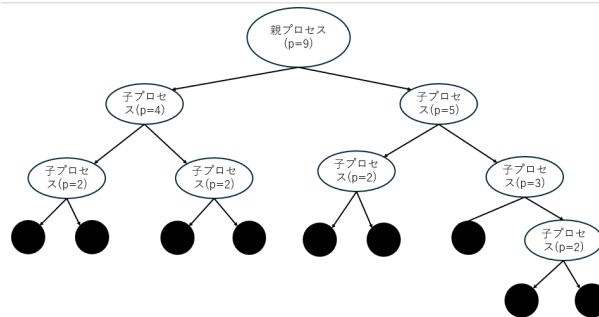


図 14: 実際にプロセスに分ける様子

7.2 実装方法

mergesort() の引数は以下のようにになっている。

```
1 void mergeSort(int numbers[], int temp[], int left, int right, int p);
```

numbers と temp は課題 3-2-2 と同じで、ソートする対象の左端と右端を表す left と right, そのプロセスにおいて作成しなければならないプロセスの個数である p を引数として追加した。先ほどのアルゴリズムで示したように、mergesort() を再帰的に呼び出す処理を行う。そのための処理としては以下のプログラムのようになる。

```
1 if (left_pid == 0) {
2     close(fd_left[0]);
3     mergeSort(numbers, temp, left, mid, p_left);
4     if (write(fd_left[1], &numbers[left], size_left * sizeof(int)) == -1) {
5         perror("pipe write.");
6         exit(1);
7     }
8     close(fd_left[1]);
9     exit(0);
10 }
```

上は左部プロセスについての処理である。再帰的に呼び出すために、mergesort() の引数として左端を left, 右端を $mid = \frac{left+right}{2}$ とした。

```
1 if (right_pid == 0) {
2     close(fd_right[0]);
3     mergeSort(numbers, temp, mid + 1, right, p_right);
4     if (write(fd_right[1], &numbers[mid + 1], size_right * sizeof(int)) == -1) {
5         perror("pipe write.");
6         exit(1);
7     }
8     close(fd_right[1]);
9     exit(0);
10 }
```

また、右部プロセスについての処理である。再帰的に呼び出すために、mergesort() の引数として左端を mid+1, 右端を right とした。パイプの処理についてはどちらのプロセスでも書き込みをしたいので二つのパイプ fd_left と fd_right を使用した。そして左部プロセスと右部プロセスの両方が終了すればそのプロセスにおける left, mid, right を使って merge() を実行する。

7.3 実行結果

以下は hatten2.c の実行結果である.

```
k-yamakb@exp029:~/enshuC/kadai3$ ./hatten2
Done with sort.
108720686
119712742
142962517
268029562
1037089053
1273908606
1297094404
1430841584
1470997468
1603641738
```

図 15: 発展課題 2 の実行結果

このことから, 正しく昇順にソートすることができていることがわかる. 課題 3-2-2 との比較については考察にて詳細を述べる.

8 発展課題 3

8.1 アルゴリズム

ここでは交互実行をセマフォを使って実装した. 具体的な処理の流れは以下のフローチャートのようなる.

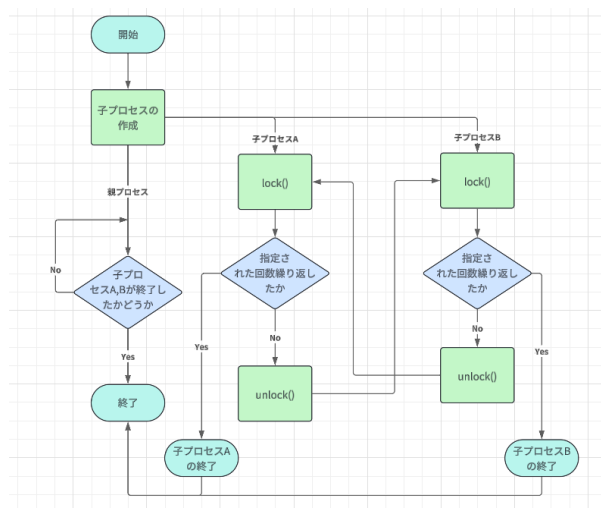


図 16: 発展課題 3 のフローチャート

ここで登場する lock() と unlock() は課題 3-1 で使用したのと同じである. unlock() から lock() に有向辺が出ているのは, セマフォの値を unlock() で増加させることで, lock() により待ち状態であったもう一方の子プロセスを開放するというを表す. これにより, お互いに待ち状態と実行状態を交互に繰り返すようにした.

8.2 実装方法

実装方法としては、以下のとおりである。

1. セマフォ用のキーを作成し、セマフォセットを作成して初期値を 1 とする。
2. 子プロセスを二つ作成する。子プロセス A ではグローバル変数の配列 A_list を要素数だけ標準出力に出力する。子プロセス B ではグローバル変数の配列 B_list を要素数だけ標準出力に出力する。
3. 親プロセスでは子プロセスの終了を待ち、両方のプロセスが終了すればセマフォを開放する。

いかに詳細を記述する。

1. ここは課題 3-1 と同様なので省略する。
2. 子プロセス A の処理は以下のようなプログラムである。

```
1      if (a_pid == 0) { // 子プロセス A
2          for (int i = 0; i < NUM; i++) {
3              lock(sem_id);
4              printf("a%d\n", A_list[i]);
5              unlock(sem_id);
6          }
7          exit(0);
8      }
```

NUM には配列 A_list と B_list の要素数を表す変数が格納されており、printf() が呼ばれるたびに lock() と unlock() が上のようなタイミングで呼び出される。これは課題 3-1 の時と同様にセマフォの値が unlock() を呼び出すたびに 1 増加されてもう一方のプロセスで待ち状態が解放されて処理が始まるということを表し、これにより交互に実行することが実現される。

また、子プロセス B の処理は以下のようなプログラムである。

```
1      if (b_pid == 0) { // 子プロセス B
2          for (int i = 0; i < NUM; i++) {
3              lock(sem_id);
4              printf("b%d\n", B_list[i]);
5              unlock(sem_id);
6          }
7          exit(0);
8      }
```

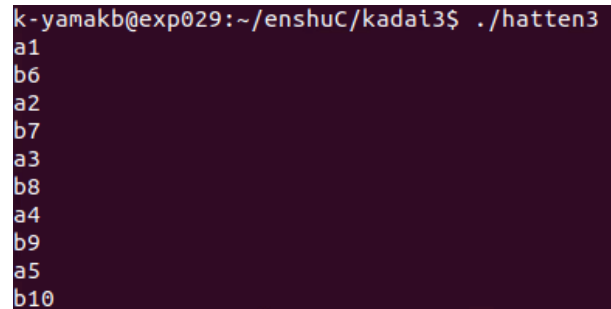
これも子プロセス A と同様である。

3. プロセスの待ちとセマフォの開放は以下ようになる。

```
1      wait(NULL);
2      wait(NULL);
3
4      // セマフォを解放
5      if (semctl(sem_id, 0, IPC_RMID) == -1) {
6          perror("semctl IPC_RMID エラー.");
7          exit(1);
8      }
```

8.3 実行結果

以下は hatten3.c の実行結果である.



```
k-yamakb@exp029:~/enshuC/kadai3$ ./hatten3
a1
b6
a2
b7
a3
b8
a4
b9
a5
b10
```

図 17: 発展課題 3 の実行結果

このことから子プロセス A と子プロセス B の処理を交互に実行することができていることがわかる.

9 考察

今回の課題を通して私が考察したのは, 発展課題 2 でも述べた通り課題 3-2-2 と発展課題 2 の実行時間の比較である. これらのプログラムの実行時間を比較するために以下のプログラムによって時間の平均を測定した.

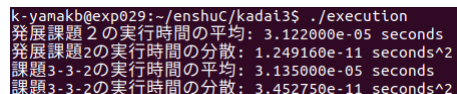
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #define NUM_TRIALS 100
6
7 double get_execution_time(const char* command) {
8     clock_t start, end;
9     start = clock();
10    system(command);
11    end = clock();
12    return ((double)(end - start)) / CLOCKS_PER_SEC;
13 }
14
15 int main() {
16     double total_time_h2c = 0.0;
17     double total_time_mc = 0.0;
18     double times_h2c[NUM_TRIALS];
19     double times_mc[NUM_TRIALS];
20
21     for (int trial = 0; trial < NUM_TRIALS; trial++) {
22         times_h2c[trial] = get_execution_time("./h2c");
23         total_time_h2c += times_h2c[trial];
24         times_mc[trial] = get_execution_time("./mc");
25         total_time_mc += times_mc[trial];
26     }
27
28     double mean_time_h2c = total_time_h2c / NUM_TRIALS;
```

```

29     double mean_time_mc = total_time_mc / NUM_TRIALS;
30
31     double variance_h2c = 0.0;
32     double variance_mc = 0.0;
33
34     for (int trial = 0; trial < NUM_TRIALS; trial++) {
35         variance_h2c += (times_h2c[trial] - mean_time_h2c) * (times_h2c[trial] -
            mean_time_h2c);
36         variance_mc += (times_mc[trial] - mean_time_mc) * (times_mc[trial] -
            mean_time_mc);
37     }
38
39     variance_h2c /= NUM_TRIALS;
40     variance_mc /= NUM_TRIALS;
41
42     printf("Average execution time for h2c: %e seconds\n", mean_time_h2c);
43     printf("Variance of execution time for h2c: %e seconds^2\n", variance_h2c);
44     printf("Average execution time for mc: %e seconds\n", mean_time_mc);
45     printf("Variance of execution time for mc: %e seconds^2\n", variance_mc);
46
47     return 0;
48 }

```

このプログラムは h2c と mc をそれぞれ 100 回ずつ実行しそれらの平均と分散を計算して表示するプログラムである。h2c は発展課題 2, mc は課題 3-3-2 の実行ファイルである。計算時間が短いので指数型表示にするようにした。また、ソートする対象の配列の要素数は多いほうが差が出やすいと考えたが、実行時間的にスムーズに比較が進められる値が 10000 個程度だったので 10000 個に設定した。また、これらのプログラムの実行はどちらも演習室環境で行った。以下はそれぞれプロセス数を変えて実行したときの結果である。

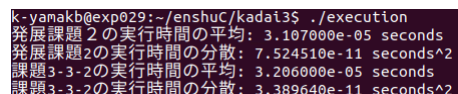


```

k-yamakb@exp029:~/enshuC/kadai3$ ./execution
発展課題 2 の実行時間の平均: 3.122000e-05 seconds
発展課題2の実行時間の分散: 1.249160e-11 seconds^2
課題3-3-2の実行時間の平均: 3.135000e-05 seconds
課題3-3-2の実行時間の分散: 3.452750e-11 seconds^2

```

図 18: プロセス数が 2 の時の実行時間



```

k-yamakb@exp029:~/enshuC/kadai3$ ./execution
発展課題 2 の実行時間の平均: 3.107000e-05 seconds
発展課題2の実行時間の分散: 7.524510e-11 seconds^2
課題3-3-2の実行時間の平均: 3.206000e-05 seconds
課題3-3-2の実行時間の分散: 3.389640e-11 seconds^2

```

図 19: プロセス数が 3 の時の実行時間

プロセス数が二つの時はどちらも同じ処理になるため平均値はほぼ同じような値となっている。ただし、分散は平均の値よりもはるかに小さく、値の大小も実行ごとに変わった。プロセス数が三つの時は並列化したほうが実行時間の平均値が小さくなっている。このようにして、プロセスの値を変えていき、それらの値をプロットしてグラフにすると以下ようになる。

参考文献

- [1] <https://www.ibm.com/docs/ja/aix/7.3?topic=f-ftok-subroutine> 6/12 アクセス
- [2] <https://www.ibm.com/docs/ja/aix/7.3?topic=s-semget-subroutine> 6/12 アクセス
- [3] <https://www.ibm.com/docs/ja/zos/2.5.0?topic=functions-wait-wait-child-process-end> 6/12 アクセス
- [4] <https://www.c-lang.net/semop/index.html> 6/12 アクセス