Vol.2021-CE-161 No.5
2021/10/23

情報工学コースのためのLinuxシェルによるコンカレント・プログラミング入門

佐藤 信[1]

概要：コンピュータおよびインターネットが重要な社会基盤となっていることから，初等・中等教育において情報リテラシおよび基本的プログラミングの学習がおこなわれるようになっている．それに対応して，大学などの情報専門コースにおけるプログラミング教育では，さらに専門性の高い内容についての学修がこれまで以上に求められるといえる．本稿では，情報工学コースでのコンカレント・プログラミング入門のための手法について述べる．その手法のために開発したモジュールを用いると，Linux のシェルを使用することによりコンカレント・プログラミングにおいて用いられるアルゴリズムについての実験が可能である．アルゴリズムの特徴を具体的に紹介するため，または，アルゴリズムの動作の要点を視覚的に提示してから詳細を説明する場合などに適した手法である．

Introduction to Concurrent Programming with Linux Shell for Information Engineering Courses

Makoto Satoh[1]

Abstract: As computers and the internet have become important social infrastructure, information literacy and fundamental programming are becoming learned in elementary and secondary education. In response, it is growing demand than ever for the programming education in information related courses at universities to learn about more specialized contents. This paper describes a method for introducing concurrent programming at information engineering courses. The modules developed for the method enable the experiments on the algorithms for concurrent programming with Linux shell. The method is suitable for introducing the specific features of the algorithms, and for illustrating their details after providing visually the essential points on their behavior.

1. はじめに

本稿では，コンカレント・プログラミングで用いられるアルゴリズムを学修するための手法[*1] [9] について説明する．手法の特徴は，次のとおりである．

- Linux のシェルからコンカレント・プログラミングのためのモジュールを組み合わせて起動する．その手順をシェル・スクリプトとして記述するとアルゴリズムの骨格に焦点をあてたプログラムを作成できる．
- アルゴリズムについての具体的なメンタル・モデルの構築を容易にするために，コンカレントな動作の可視化をおこなう．

この手法は情報工学コースなどにおいてコンカレント・プログラミングで用いられるアルゴリズムの特徴を具体的に紹介するため，または，コンカレント・プログラミング入門においてアルゴリズムの要点を視覚的に分かりやすく提示してから詳細を説明する場合に適した手法である．

これ以降の構成について説明する．2 節では，関連研究と本稿で用いる手法の比較をおこなう．Linux のシェルを

[1] 岩手大学
Iwate University, Ueda, Iwate 020–8551, Japan

[*1] 提案手法の実装を公開している．
https://blue0.an.cis.iwate-u.ac.jp/ConcurrentProgrammingToolkit/

使用したコンカレント・プログラミング手法について，3 節において述べる．4 節ではシェルを用いたコンカレント・プログラミングの実験例を示し，検討をおこなう．最後の 5 節において本稿のまとめと今後について述べる．

2. 関連研究

2.1 コンカレント・プログラミングとは

コンカレント・プログラミングとは，複数のプログラムを連携動作させる場合などに必要となるプログラミングの手法である [2], [10] ．もともとは，マルチプログラミングのオペレーティング・システム (OS) の研究・開発のために必要な技術として研究・開発が始められた手法である．現在の OS (Linux, Windows, macOS など) にも必要不可欠な技術である．そして，次に示すようなプログラミングにおいてもよく用いられる技術である．

- 複数のプログラムによる共有データの更新
- 複数のプログラムの同期
- ユーザ・インタフェース
- ネットワーク経由での分散プログラミング

これらの技術は日常的に使用されているソフトウエアにおいて用いられていることから，標準的な PC ユーザであればコンカレント・プログラミングの技術の恩恵を受けているといえる．また，インターネットではネットワークを経由して複数のプログラム間で通信がおこなわれているということに着目すると，壮大な規模でのコンカレント・プログラミングがおこなわれている空間としてインターネットを捉えることも可能である．

コンカレント・プログラミングでは，複数個のプロセスまたはスレッド (以降では，エンティティと表記) を生成し，アルゴリズムによりエンティティの動作の関連づけをおこなう．そのアルゴリズムを学修する準備として，次を学修している必要がある．

- 複数個のエンティティの生成
- エンティティの関連動作のためのプロセス間通信[*2]

コンカレント・プログラミングをおこなう場合には，そのための機能をプログラミング言語の仕様として備えている言語 (Java, C++など)，または，プロセス間通信のためのライブラリを用いる．それらの内部では OS が提供するコンカレント・プログラミングのための機能を使用していることも多いので，それについても習熟する必要がある．

[*2] スレッド間での通信についても，プロセス間通信という用語を用いることが多い．また，プロセス間通信という用語は，プロセス間でデータを授受する場合だけではなく，プロセスがシステムの機能を用いて間接的に別のプロセスの動作を制御する場合にも用いられる．

本稿で用いる手法では，Linux シェルからコンカレント・プログラミングのためのモジュールを起動することによりアルゴリズムの実験をおこなう．必要となるのは，シェルに関する入門レベルの知識である．シェル・スクリプトは変更が容易であるので，アルゴリズムを実行するエンティティの個数の変更をおこないながらアルゴリズムの動作を確認するような試行錯誤による実験が容易である．

2.2 コンカレント・プログラミング入門

コンピュータおよびインターネットは重要な社会基盤であることから，初等・中等教育において情報リテラシおよび基本的なプログラミングについての学習がおこなわれるようになっている．それに対応して，大学などの情報専門コースにおけるプログラミング教育では，さらに専門性の高い内容についての学修がこれまで以上に求められる．

情報専門コースでのプログラミング教育について多くの研究結果が発表されている [8] ．情報工学コースにおいてプログラミングの基礎の次に学修する内容には基本的なコンピュータ・アルゴリズムなどがあり，標準的なカリキュラムに含まれている．より高度な内容として学修するべき内容には，コンカレント・プログラミング [2], [6], [10] を候補のひとつとしてあげることができる．2.1 節で述べたように，PC などの標準的なユーザであればコンカレント・プログラミングの技術の恩恵を日常的に受けている．一方では，コンカレントな動作をおこなうプログラムの作成では専門的なアルゴリズムの知識が必要となる．そのため，コンカレント・プログラミングに関する講義または実習が情報専門コースのカリキュラムに含まれる場合がある [4], [7] ．また，そのアルゴリズムではエンティティ間で通信をおこなうことによりエンティティの動作の関連づけをおこなうことから，アルゴリズムの理解を容易にするためには可視化が有効であり，そのための研究が多数おこなわれている [1], [5], [11] ．

情報科学の入門コースにおいてコンカレント・プログラミングを取り扱った研究には [3] がある．プログラミング言語 (Java) にあるコンカレント・プログラミングのための機能を用いたクラスを予め準備しておいて，そのクラスを継承したクラスを作成することにより実験をおこなっている．アルゴリズムの各ステップを学修するというよりは，コンカレント・プログラミングとはどのようなものなのかをプログラムのグラフィカルな動作により体験している．履修者へのアンケートでは，コンカレント・プログラミングに関する題材については，難しい (他の題材と比較して difficulty のスコアが高い) が面白くて (fun が高い) 学修する価値がある (educational value が特に高い) という回答が多く得られたことが報告されている．

本稿で用いる手法では，アルゴリズムの骨格に焦点をあてた実験が可能である．シェル・スクリプトにより実験をおこないながら，アルゴリズムの各ステップを次のように確認できる．

- 各モジュールからの端末への出力
- 各モジュールの動作を可視化したグラフ

また，2.1 節で述べたようなアルゴリズムを学修するための準備を必要とせずに，シェル・スクリプトの入門レベルの知識によりアルゴリズムの実験が可能である．

3. Linux のシェルを使用したコンカレント・プログラミング

3.1 アトミシティの実験のためのモジュール

ある一連の動作が 1 つのエンティティのみによりおこなわれる場合に，アトミシティ (Atomicity, 原子性) が確保されているという．図 1 (下側) により具体的に説明する．ここでは，プロセス[*3] **P1** および **P2** が，共有変数[*4] **sum** の値を更新している．**P1** の動作は次のとおりである．

Step 1 プロセス **P1** は，共有変数 **sum** の値を **P1** 内の変数に読み込む ($\texttt{get}_{(\mathbf{P1})}$).

Step 2 **P1** は，読み込んだ値に 1 を加算してから加算結果を **sum** に書き込む ($\texttt{put}_{(\mathbf{P1})}$).

プロセス **P2** についても同様である．**sum** の初期値が 0 であるとすると，動作の順序により次のような結果となる．

順序パターン 1 (加算結果 1):

$$\texttt{get}_{(\mathbf{P1})} \Rightarrow \texttt{get}_{(\mathbf{P2})} \Rightarrow \texttt{put}_{(\mathbf{P1})} \Rightarrow \texttt{put}_{(\mathbf{P2})}$$

順序パターン 2 (加算結果 2):

$$\texttt{get}_{(\mathbf{P1})} \Rightarrow \texttt{put}_{(\mathbf{P1})} \Rightarrow \texttt{get}_{(\mathbf{P2})} \Rightarrow \texttt{put}_{(\mathbf{P2})}$$

可能な動作の順序には，他にも複数の組み合わせがある．各プロセスは OS により CPU を割り当てられて実行されるが，どのプロセスをどれだけの時間 CPU に割り当てるのかはシステムの状態などを基に OS が決定する．**P1** の動作のみに着目すると，

$$\texttt{get}_{(\mathbf{P1})} \Rightarrow \texttt{put}_{(\mathbf{P1})}$$

という順序は保証される．一方，$\texttt{get}_{(\mathbf{P2})}$ の動作については，$\texttt{get}_{(\mathbf{P1})}$ より前，$\texttt{get}_{(\mathbf{P1})}$ と $\texttt{put}_{(\mathbf{P1})}$ の間，あるいは，$\texttt{put}_{(\mathbf{P1})}$ の後のいずれになるのかは，OS によるプロセスの割り当てがどのようにおこなわれるのかによる．$\texttt{put}_{(\mathbf{P2})}$ についても同様である．

[*3] プログラムを起動するとプロセスが生成され，プロセスによりプログラムに記述した処理がおこなわれる．
[*4] 共有変数は，プロセスの外部に確保された変数である．

表 1: アトミシティの実験のためのモジュール
Table 1 Modules for the experiments of atomicity.

modules	description
StartCP	Preparing a shared variable.
Add1	Adding 1 to the shared variable.
Print	Verifying the shared variable.
EndCP	Removing the shared variable.
ProcessBehaviorGraph	Generating visualization.

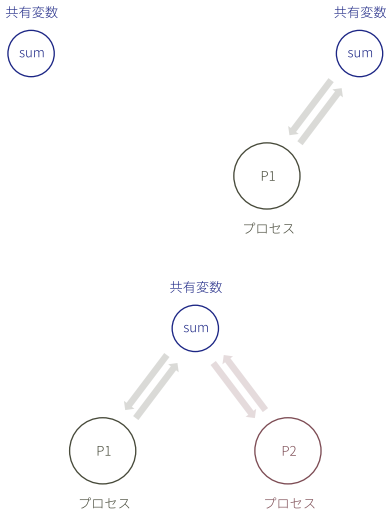

図 1: 各モジュールの動作
Fig. 1 Behavior of Modules: StartCP (top left), and adding by 1 with single Add1 module (top right) and with multiple Add1 modules (bottom).

コンカレント・プログラミングためのアルゴリズムを用いると，プロセスを必ず**順序パターン 2** で動作させることが可能である．その場合には，プロセスによる共有変数の更新においてアトミシティが確保されているという．

表 1 に，Atomicity の実験用のモジュールを示す．

`StartCP` 共有メモリ領域を確保し，その領域に共有変数 **sum** を用意する．

`Add1` `Add1` プロセスを 1 つ生成する．`Add1` プロセスは，共有変数 **sum** に格納されている値を読み込み，その値に 1 を加算する．そして，加算結果で **sum** の値を更新する (オプションでアトミシティを指定).

`Print` 共有変数 **sum** に格納されている値を表示する．

`EndCP` 共有メモリ領域を解放する (システムに返却する).

`ProcessBehaviorGraph` プロセスの動作を可視化する．

図 1 は，各モジュール動作である．

表 2: エンティティの同期の実験のためのモジュール
Table 2 Modules for the experiments of synchromization.

modules	description
StartCP	Preparing shared variables.
SetTotalWorkers	Setting total sorting workers.
SetValues	Setting sorting values.
Sort	Launching a sort process.
Merge	Launching a merge process.
EndCP	Removing the shared variables.

3.2 エンティティの同期の実験のためのモジュール

複数のエンティティの動作のタイミングをあわせることを，エンティティを同期 (Synchronization) させるという．ここでは，複数のプロセスが作業を分担することによりマージ・ソート[*5] をおこなう．複数のソート専用プロセスが分担して部分的なソートをおこない，それが終了するのを待ちマージ専用プロセスがマージをおこなう．そのためのモジュールを，表 2 に示す．

`StartCP` 共有メモリ領域を確保し，その領域に共有変数を用意する．
`SetTotalWorkers` ソート・プロセスの総数を指定する．
`SetValues` ソートする数列を生成する．
`Sort` ソート・プロセスを 1 つ生成する．
`Merge` マージ・プロセスを 1 つ生成する．
`EndCP` 共有メモリ領域を解放する (システムに返却する)．共有変数も解放される．

マージ・ソートの手順は，次のとおりである．

部分数列の生成 `SetValues` モジュールにより数列を生成する．生成した数列を，`SetTotalWorkers` モジュールで指定したソート・プロセスの総数で分割することにより部分数列を生成する．
ソート 生成した部分数列を，`Sort` モジュールにより生成した複数のソート・プロセスが分担してソートする．
マージ ソートした部分数列を，`Merge` モジュールにより生成したマージ・プロセスがマージする．

プロセスの動作をセマフォを用いて制御するかどうかは、モジュールのオプションで指定する．セマフォを使用すると，`Sort` モジュールと `Merge` モジュールの起動順序に依存せずにマージ・ソートをおこなえる．

[*5] コンピュータ・アルゴリズムの分野でマージ・ソートという場合には，複数要素からなる数列を 2 要素からなる部分数列に分割してから，ソートおよびマージを繰り返す．ここでは、コンカレントな動作の実験を簡単するために，全要素をソート・プロセスの総数で分割した部分数列を用いる．

4. 実験と結果の検討

4.1 実装

Linux のシェル・スクリプトによりコンカレント・プログラミングの実験をおこなうために，3 節でのモジュールを実装した．

実装に使用した言語は C++である．実験をおこなう場合には，作成したモジュールをプログラムとしてシェルから起動する．シェルのプロンプトからモジュールを起動，または，モジュールを起動する手順を記述したシェル・スクリプトによりモジュールを起動することが可能である．

Atomicity の実験のためのモジュールでは，共有変数の更新でのアトミシティを確保するために名前付きセマフォを使用した．エンティティの同期の実験のためのモジュールでは，共有変数の配列に数列を格納した．プロセスの動作の制御のために，2 つの名前付きセマフォを使用した．全ソート・プロセスがソートを完了したことをあるソート・プロセスが検出するためのセマフォ，および，全ソート・プロセスがソートを完了したことをマージ・プロセスが検出するためのセマフォを使用した．名前付きセマフォを用いたのは，複数のセマフォを管理するため，および，Linux などで複数のユーザが本モジュールを使用して実験をおこなうような場合に対応するためである．あるユーザが発生させたエラーが他のユーザの実験に影響を及ぼさないようにモジュールを設計している．

4.2 対話的入力による実験

シェルのプロンプトに対話的にモジュール名を入力することにより実験をおこなった例を，図 2 に示す．`StartCP` モジュールにより共有変数を準備し，`Add1` モジュールにより共有変数の値に 1 を加算し，`Print` モジュールにより共有変数の値を確認し，そして，`EndCP` モジュールにより共有変数を解放している．`Add1` モジュールからは，共有変数から読みだした値，および，共有変数に書き出した値が出力されている．

```
$ ./StartCP
$ ./Add1
add1 process 1: get value: 0
add1 process 1: set value 1
$ ./Print
Shared memory value: 1
$ ./EndCP
```

図 2: 対話的入力による実験
Fig. 2 Experiment with interactive inputs.

```
$ ./StartCP
$ ./Add1 --semaphore false 1 & \
  ./Add1 --semaphore false 2 &
add1 process 1: get value: 0
add1 process 2: get value: 0
add1 process 1: set value 1
add1 process 2: set value 1
$ ./Print
Shared memory value: 1
```

図 3: セマフォを使用しない動作の例
Fig. 3 Example of the behavior without a semaphore.

```
$ ./StartCP
$ ./Add1 --semaphore true 1 & \
  ./Add1 --semaphore true 2 &
add1 process 1: get value: 0
add1 process 1: set value 1
add1 process 2: get value: 1
add1 process 2: set value 2
$ ./Print
Shared memory value: 2
```

図 4: セマフォを使用する動作の例
Fig. 4 Example of the behavior with a semaphore.

図 3 では，Add1 モジュールを 2 つ起動し，2 つの Add1 プロセスにより加算をおこなっている．セマフォを使用してないので，共有変数の更新ではアトミシティが確保されずに加算結果は 1 または 2 になる可能性がある．この場合には，共有変数に格納されている加算結果は 1 となった．図 4 では，セマフォを使用して共有変数の更新でのアトミシティを確保しているので，加算結果は 2 となった．

4.3 シェル・スクリプトによるアトミシティの実験

リスト 1 は，アトミシティの実験をおこなうためのシェル・スクリプトである．Linux の bash のシェル・スクリプトにより，StartCP, Add1, Print, EndCP モジュール (茶色) を起動している．Add1 モジュールでは，オプション --semaphore true によりセマフォを使用することを指定している (緑色)．セマフォを使用しない場合には，false に変更する．$count では，Add1 モジュールの ID を指定している．また，可視化のためのプロセスの動作ログを収集するために，StartCP, Add1, EndCP モジュールではオプション -l を指定している (青色)．図 5 では，リスト 1 のシェル・スクリプト Sum.sh を起動し，ProcessBehaviorGraph モジュールで可視化したグラフを生成している．図 6 は，./Sum.sh で --semaphore false としたプロセスの動作の可視化である．図 7 は，--semaphore true とした可視化である．

リスト 1: アトミシティの実験のためのシェル・スクリプト
Listing 1: Shell script for the experiment of atomicity.

```
#!/bin/bash
./StartCP -l
for (( count = 1; count <= $1; count ++ )) do
  ./Add1 --semaphore true -l $count &
done
wait
./Print
./EndCP -l
```

```
$ ./Sum.sh 2
add1 process 2: get value: 0
add1 process 2: set value 1
add1 process 1: get value: 1
add1 process 1: set value 2
Shared memory value: 2
$ ./ProcessBehaviorGraph
```

図 5: ログの収集および可視化の生成
Fig. 5 Collecting the process log and generating visualization.

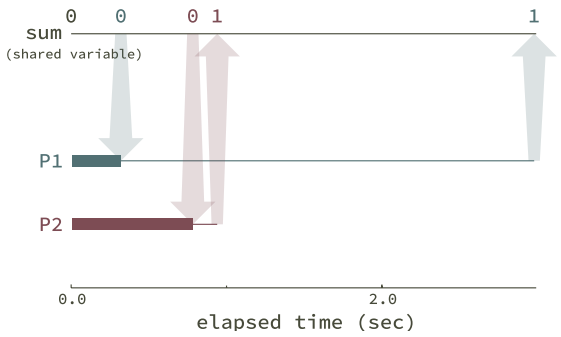

図 6: アトミシティの実験の可視化 (セマフォ使用せず)
Fig. 6 Visualizing the experiments of atomicity (without semaphore).

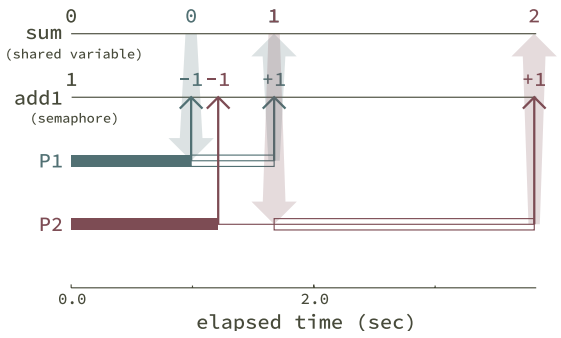

図 7: アトミシティの実験の可視化 (セマフォ使用)
Fig. 7 Visualizing the experiments of atomicity (with semaphore).

```
$ ./StartCP
$ ./SetTotalWorkers 2
$ ./SetValues 10
$ ./Merge &
$ ./Sort 1 & ./Sort 2 &
sorting process 2: sorting values 5 9 3 2 4
sorting process 2: sorted values 2 3 4 5 9
sorting process 1: sorting values 5 8 4 3 10
sorting process 1: sorted values 3 4 5 8 10

merging process: merged values 2 3 3 4 4 5 5 8 9 10
```

図 8: 対話的なエンティティの同期 (セマフォを使用)
Fig. 8 Interactive entity synchronization (with semaphore).

```
$ ./StartCP
$ ./SetTotalWorkers 2
$ ./SetValues 10
$ ./Merge --semaphore false &
merging process: partially sorted values of \
sort process 1 do not found !
```

図 9: セマフォを用いないエンティティの動作
Fig. 9 Entity behaviors without semaphore.

4.4 エンティティの同期の実験

図 8 は，シェルのプロンプトに対話的にモジュール名を入力することによりエンティティの同期についての実験をおこなった様子である．

始めに，StartCP モジュールにより，共有変数のためのメモリ領域を準備している．そこには，ソートする数列，部分数列，および，ソートした数列などを格納する．次に，SetTotalWorkers モジュールによりソート・プロセスの数 (2) を設定している．そして，SetValues モジュールにより要素数 $N(10)$ の数列を生成している．数列の要素は，$[1, N]$ から重複を許して乱数により選択した正整数である．数列の要素が重複しているのは，数列の長さからソートした数列を容易に判断できないようにするためである．

次に, Merge モジュールを起動し, バックグラウンド・プロセスとして Merge プロセスを生成している．--semaphore オプションを指定しない場合の規定値は true である．Merge プロセスは開始するが，全部分ソートの終了の待ち状態に入る．

そして，Sort モジュールを 2 つ起動し，バックグラウンド・プロセスとして Sort プロセスを 2 つ生成している．Sort モジュールの引数には ID(1 または 2) を指定している．ID を指定するのは，複数の Sort プロセスからの出力を区別するためである．--semaphore オプションを指定しない場合の規定値は true である．Sort プロセスは，部分数列のソートを開始する．全 Sort プロセスが部分ソートを完了すると，Merge モジュールがソート済みの部分数列をマージし共有領域に格納する．全要素をソートした数列が，端末に出力されている．

リスト 2: エンティティの同期の実験のためのシェル・スクリプト
Listing 2: Shell script of the experiment of synchronization

```
#!/bin/bash
./StartCP
./SetTotalWorkers $1
./SetValues $2
for (( count = 1; count <= $1; count ++ )) do
  ./Sort --semaphore true  $count &
done
./Merge --semaphore true
./EndCP
```

```
$ ./MergeSortWithSemaphore.sh 2 10
sorting process 1: sorting values 6 3 1 8 7
sorting process 1: sorted values 1 3 6 7 8
sorting process 2: sorting values 3 7 2 2 5
sorting process 2: sorted values 2 2 3 5 7

merging process: merged values 1 2 2 3 3 5 6 7 7 8
```

図 10: リスト 2 を用いたエンティティの同期
Fig. 10 Entity synchronization with **Listing 2**.

図 9 では，オプション--semaphore false を指定して Merge モジュールを起動している．Sort プロセスが全部分ソートを完了するのを待たずに (同期をとらずに)，Merge プロセスがソート済みの部分数列を読み込もうとするので，エラーが発生している．

図 8 と同様の実験をシェル・スクリプトを用いておこなった．リスト 2 は，そのシェル・スクリプトである．シェル・スクリプトの引数により，ソート・プロセス数 ($1) およびソートする数列の要素数 ($2) を指定できるようにしている．図 10 は，リスト 2 のシェル・スクリプトの実行結果である．2 つの Sort プロセスが分担して部分数列のソートをおこない，Merge プロセスがソート済みの部分数列をマージしてソート結果を出力している．

セマフォを用いている実験では，2 つのセマフォを使用した．ある Sort プロセスが部分ソートを完了した時点で全 Sort プロセスがソートを完了しているかどうかを確認するためのセマフォ，および，全 Sort プロセスの部分ソートの完了を Merge プロセスが待つためのセマフォである．

4.5 検討

3 節で説明したコンカレント・プログラミングのためのモジュールを用いて実験をおこなった．

4.2 節では，シェルのプロンプトに対してモジュール名を入力することにより，コンカレント・プログラミングの対話的な実験が可能であることを確認できた．図 2 では，StartCP モジュールで確保した共有変数の値を Add1 モジュールにより更新した．Add1 プロセスの出力からは，共有変数の値を読み出し，その値に 1 を加算した値を共有変数に書き出す過程を確認できる．更新後の共有変数の値を，Print モジュールの出力から確認できる．図 3 では，2 つの Add1 プロセスにより共有変数の更新をおこなった．この実験では，プロセスによる共有変数の更新でのアトミシティを確保していない．この場合には，加算結果が 1 になった．アトミシティを確保しない場合にはプロセスによる共有変数の操作の組み合わせは複数存在する (3.1 節を参照) ので．図 3 の加算結果が 2 になる場合もある．図 4 では，セマフォを使用して Add1 プロセスによる共有変数の更新でのアトミシティを確保した．図 3 と同様の実験をおこなっているが，この場合には加算結果が 2 になることが確認できる．アトミシティを確保しているので，この場合には，実験を繰り返しても必ず加算結果は 2 になる．

4.3 節のリスト 1 は，前節での実験を自動化するためのシェル・スクリプトである．シェル・スクリプトの引数に生成する Add1 プロセス数を指定できるようにした．また，プロセスの動作を可視化するためのログを収集するためのオプションをモジュールに指定している．図 5 では，リスト 1 を実行することによりログを収集し，ログからグラフを自動生成している．図 6 では，セマフォを用いない場合のプロセスの動作の可視化をおこなっている．ID1 の Add1 プロセス (P1) および ID2 の Add1 プロセス (P2) が共有変数 **sum** の値を更新するための操作の順序が，

$$\mathtt{get}_{(\mathbf{P1})} \Rightarrow \mathtt{get}_{(\mathbf{P2})} \Rightarrow \mathtt{put}_{(\mathbf{P2})} \Rightarrow \mathtt{put}_{(\mathbf{P1})}$$

であることを確認できる．図 7 では，セマフォを使用して共有変数のアトミシティを確保している．プロセスが名前付きセマフォ (add1) を操作することにより，プロセスの動作を制御しながらアトミシティを確保していることを確認できる．この場合には，

$$\mathtt{get}_{(\mathbf{P1})} \Rightarrow \mathtt{put}_{(\mathbf{P1})} \Rightarrow \mathtt{get}_{(\mathbf{P2})} \Rightarrow \mathtt{put}_{(\mathbf{P2})}$$

である．

図 12 は, 図 11 でのアトミシティの実験を可視化したものである．図 11 では，モジュールのオプションに--verbose を指定することによりプロセスの動作の詳細な情報を出力

```
$ ./Sum-Verbose.sh 2
add1 process 2: sleep 312 milliseconds.
add1 process 1: sleep 598 milliseconds.
add1 process 2: decrement semaphore for atomic addition.
add1 process 2: starting critical section.
add1 process 2: get value: 0
add1 process 2: sleep 1310 milliseconds.
add1 process 1: decrement semaphore for atomic addition.
add1 process 2: set value 1
add1 process 2: increment semaphore for atomic addition.
add1 process 1: starting critical section.
add1 process 2: Ending critical section.
add1 process 1: get value: 1
add1 process 1: sleep 627 milliseconds.
add1 process 1: set value 2
add1 process 1: increment semaphore for atomic addition.
add1 process 1: Ending critical section.
Shared memory value: 2
$ ./ProcessBehaviorGraph
```

図 11: プロセスの動作の詳細
Fig. 11 Details of process behavior.

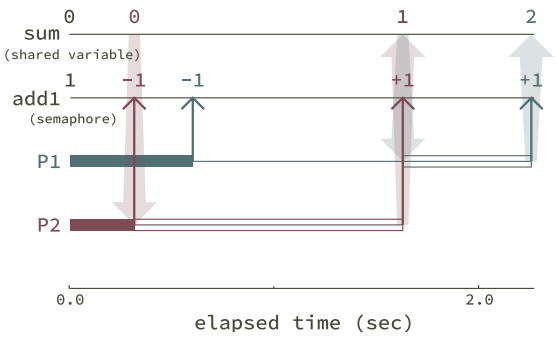

図 12: 図 11 の可視化
Fig. 12 Visualization of Fig.11.

している．その出力情報と図 12 を比較すると，プロセスの動作の可視化がアルゴリズムの動作の把握に有効であることが分かる．複数のプロセスが相互作用するようなコンカレント・プログラミングのためのアルゴリズムについてのメンタル・モデルの構築を容易にできるといえる．

4.4 節では，エンティティの同期についての実験をおこなった．図 8 からは，セマフォを用いることにより，Sort プロセスよりも先に Merge プロセスを生成しても，全 Sort プロセスが部分ソートを完了してから Merge プロセスがマージをおこなうように，プロセスの動作を制御できることを確認できる．図 9 からは，同様の操作をセマフォを用いずにおこなおうとすると，Sort プロセスの部分ソートの完了を待たないで Merge プロセスがマージをおこなおうとするためにエラーが発生することを確認できる．リスト

2 は，同様の実験をおこなうためのシェル・スクリプトである．図 10 は，その実行結果である．シェル・スクリプトによりモジュールを起動しても，図 8 と同様の実験が可能であることが分かる．シェル・スクリプトの引数に指定するソート・プロセス数およびソートする数列の要素数を変更することにより，実験の試行錯誤などが容易である．

5. おわりに

本稿では，コンカレント・プログラミングにおいて用いられるアルゴリズムを学修するための手法について述べた．特徴は，Linux のシェルによりアルゴリズムの実験をおこなえる点である．シェルにはないコンカレント・プログラミングのための機能については，専用のモジュールにより実現した．シェル・スクリプトを作成することにより試行錯誤による実験が容易である．また，プロセスの動作の可視化を自動的におこなえるので，アルゴリズムの具体的なメンタル・モデルの構築を容易にすることができる．情報工学コースなどにおいてコンカレント・プログラミングで用いられるアルゴリズムの特徴を具体的に紹介するため，または，コンカレント・プログラミング入門においてアルゴリズムの要点を視覚的に分かりやすく提示してから詳細を説明する場合などに適した手法である．

今後の課題には，各種のアルゴリズムに対応するためのモジュールの改良，および，より分かりやすい可視化手法に関する研究などがある．

参考文献

[1] Adams, J. C., Koning, E. R. and Hazlett, C. D.: Visualizing Classic Synchronization Problems: Dining Philosophers, Producers-Consumers, and Readers-Writers, SIGCSE '19, New York, NY, USA, ACM, pp. 934–940 (2019).
[2] Andrews, G. R. and Schneider, F. B.: Concepts and Notations for Concurrent Programming, *ACM Comput. Surv.*, Vol. 15, No. 1, pp. 3–43 (1983).
[3] Bruce, K. B., Danyluk, A. and Murtagh, T.: Introducing Concurrency in CS 1, SIGCSE '10, ACM, pp. 224–228 (online), DOI: 10.1145/1734263.1734341 (2010).
[4] Ernst, D. J. and Stevenson, D. E.: Concurrent CS: Preparing Students for a Multicore World, *SIGCSE Bull.*, Vol. 40, No. 3, p. 230?234 (2008).
[5] Grissom, S., McNally, M. F. and Naps, T.: Algorithm Visualization in CS Education: Comparing Levels of Student Engagement, *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, ACM, pp. 87–94 (2003).
[6] Hansen, P. B.: Concurrent Programming Concepts, *ACM Comput. Surv.*, Vol. 5, No. 4, pp. 223–245 (1973).
[7] Jackson, D.: A Mini-Course on Concurrency, *SIGCSE*, Vol. 23, No. 1, pp. 92–96 (1991).
[8] Luxton-Reilly et al.: Introductory Programming: A Systematic Literature Review, ITiCSE 2018 Companion, ACM, pp. 55–106 (online), DOI: 10.1145/3293881.3295779 (2018).
[9] 佐藤　信：Linux シェルを用いたコンカレント・プログラミング入門，2021 年度電気関係学会東北支部連合大会予稿集，p. 4C02 (2021).
[10] Park, D. A.: Concurrent Programming in a Nutshell, *J. Comput. Sci. Coll.*, Vol. 23, No. 4, pp. 51–57 (2008).
[11] Shaffer, C. A., Cooper, M. L., Alon, A. J. D., Akbar, M., Stewart, M., Ponce, S. and Edwards, S. H.: Algorithm Visualization: The State of the Field, *ACM Trans. Comput. Educ.*, Vol. 10, No. 3, pp. 1–22 (2010).