UNIXシステムプログラミング

2019年度版

慶應義塾大学理工学部情報工学科
寺岡文男

第1章　はじめに

1.1　本科目のねらい

本科目は情報工学科3年生を対象としている．皆さんは来年4月から研究室に所属して卒業研究を行うことになる．研究分野によって程度の差はあると思うが，多くの学生は卒業研究においてなんらかのプログラミングをすることになるであろう．現在の情報工学科では，プログラミングとしては2年から3年の春学期までにC言語とJavaを学習している．プログラミングの科目においては文法の理解に重点が置かれ，実際のプログラミングとしては比較的簡単な練習問題を多数解くという形式である．1つの練習問題におけるプログラム量は多くても40〜50行程度ではないかと思う．

しかし，卒業研究のために行うプログラミングは練習問題を解くためのプログラミングとは量・質の両面で大きな違いがある．特にコンピュータサイエンス系の研究室では，たとえばクライアント・サーバシステムの全体をプログラミングしたり，あるいはハードウェアを制御するためのプログラミングをしたり，または測定のためのプログラミングをしたりする．数百行から千行を超えるプログラミングを行うことはまれではない．プログラミング言語の文法を覚え練習問題を解くだけのプログラミング能力と，卒業研究で必要となるプログラミング能力の間に大きなギャップがある．このギャップを埋めるのが本科目のねらいである．本科目はタイトルのとおりUNIX上のC言語によるシステムプログラミング技術を習得する．併せて，プログラムの実行環境を提供しているUNIXオペレーティングシステムの動作の一部も理解する．

2019年度に関して　本科目は2006年度に新しく設置された選択科目である．本科目は1時限のみであり，演習のための明確な時間は取られていない．ある日は1時限を通して講義のみだったり，ある日は1時間程度が講義で残り時間を簡単な演習に当てるかもしれない．成績評価は5題の課題によって行い，期末試験は行わない．本講義を通じて，システムプログラミングの面白さを知って欲しい．

本資料は2006年度に書き下ろしたものであり，毎年これに加筆して用いている．間違い等についてはその都度修正しているが，間違いが残っていたり，加筆部分に間違いがあるかもしれない．そのような箇所を見つけたらぜひ知らせて欲しい．

実は本科目は来年度から4年春学期の科目になる．したがって，3年秋学期としての講義は今年が最後である．来年度(2020年度)春学期は休講となり，2021年度春学期から4年生用の科目として内容を一新する予定である．

図 1.1: オペレーティングシステムとシステムソフトウェア

1.2 前提科目

UNIX システムプログラミングは情報工学科 3 年生を対象として 2006 年度から秋学期に設置された選択科目である．情報工学科では，2 年春学期に「プログラミング第 1 同演習」(必修科目) が設置され，C 言語を用いたプログラミングの基礎を習得する．さらに 2 年秋学期に「プログラミング第 3 同演習」(選択科目) が設置され，C 言語の文法全般を理解する．本科目はプログラミング第 3 同演習が履修済みであるか，同程度の C 言語の知識を有していることを前提とする．

本科目にはオペレーティングシステムの基礎知識が必要となる部分もあるので，本科目と同時期 (3 年秋学期) に設置されている「オペレーティングシステム」も同時に履修していることが望ましい．また，ネットワークプログラミングも扱うので，3 年春学期に設置されている「ネットワーク工学 I」が履修済みであるか，3 年秋学期に設置されている「ネットワーク工学 II」を同時に履修していることが望ましい．

1.3 システムソフトウェアとは

図 1.1 にオペレーティングシステムとシステムソフトウェアの関係を示す．一番下にハードウェアが位置し，そのすぐ上にはオペレーティングシステムのカーネル部分が位置する．カーネルはコンピュータが動作している間メインメモリに常駐し，すべてのプロセス[1]で共有される．狭義のオペレーティングシステムはこのカーネル部分を指すことが多い．カーネルの上にはミドルウェアやサーバプログラムなど，ユーザが直接接することがないソフトウェアが位置する．たとえば Web サーバやメールサーバなどである．通常，システムソフトウェアとはオペレーティングシステムのカーネル部分からミドルウェアやサーバ類までを指すことが多いが，狭義ではミドルウェアやサーバ類のみを指すこともある．一番上位にはアプリケーションプログラムが位置する．本科目では狭義のシステムソフトウェアの部分を対象とする．

オペレーティングシステムの重要な機能の 1 つはコンピュータの資源を管理することである．コンピュータの資源とは，処理装置である CPU，記憶装置であるメインメモリやハードディスク，入出力装置であるキーボード，ディスプレイ，ネットワークなどである．これらのうち，CPU や入出力装置は使用権が時間的に分割されて管理されている．CPU

[1] プロセスの定義は第 1.4 節を参照．

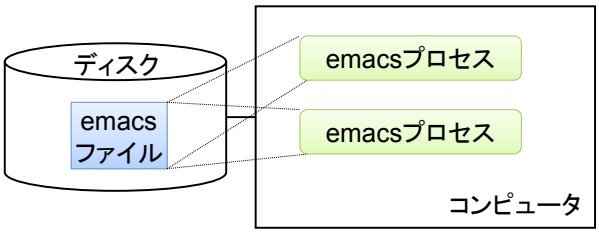

図 1.2: プログラムとプロセス

や入出力装置はある一時点においては 1 つのプロセスに占有されるため，オペレーティングシステムはこのような資源の使用権を時間的に分割し，各プロセスに配分する．一方，メインメモリやハードディスクの使用権は空間的に分割されて管理されている．

オペレーティングシステムのもう 1 つの重要な機能は，アプリケーションプログラムに対してハードウェアの違いを隠蔽し，共通した API (Application Programming Interface) を提供することである．大規模なサーバコンピュータ，身近に使用しているデスクトップ PC やノート PC，さらにスマートホン (スマホ) など，コンピュータによって使用しているハードウェア (たとえば CPU，メモリ，ディスク，ネットワークカード) はさまざまである．しかし共通のオペレーティングシステムを利用すれば，このようなハードウェアの違いを意識せずに共通のアプリケーションプログラムを作成することができる．

オペレーティングシステムのカーネル部分が API として提供するのは比較的低機能なサービスである．高度なアプリケーションプログラムを作成するには，オペレーティングシステムが直接提供するサービスだけでは不十分な場合が多い．そこでオペレーティングシステムのカーネル部分の上で，さらに高機能なサービスをアプリケーションプログラムに提供するためにミドルウェアやサーバなどが動作する．広義のオペレーティングシステムはこの部分も含むことが多い．

1.4 プログラムとプロセス

プログラムとはコンピュータに対する命令やデータの集まりであり，通常はハードディスク上にファイルとして格納されている．たとえば UNIX の “`ls`” というコマンドは，“`/bin/ls`” という実行形式のファイルとして格納されている．一方，プロセスとは簡単に言うと実行状態のプログラムのことである．カーネルの観点からは，CPU 時間やメモリ領域などの資源はプロセスを対象として割り当てられる．たとえばユーザが `ls` の実行を開始すると新しくプロセスが生成され，そのプロセスが `ls` というプログラムを実行する．`ls` というプログラムを実行しているプロセスを “`ls` プロセス” と呼ぶことが多い．同一のプログラムを同時に複数実行すると，その分だけプロセスが生成される．たとえば，図 1.2 に示すように `emacs` を同時に 2 つ実行すると，`emacs` プロセスが 2 つ生成される．

各プロセスは仮想メモリ空間[2]を持つ．32 ビットの CPU の場合，各プロセスは `0x00000000` 番地から `0xFFFFFFFF` 番地までの仮想メモリ空間を持つ．多くの UNIX システムでは，高位番地 (たとえば `0x80000000` から上位) をカーネル空間が占める．カーネル空間はすべてのプロセスで共有される．図 1.3 にプロセスの仮想メモリ空間構造を示す．この図では，

[2]仮想メモリについてはオペレーティングシステムの講義参照．

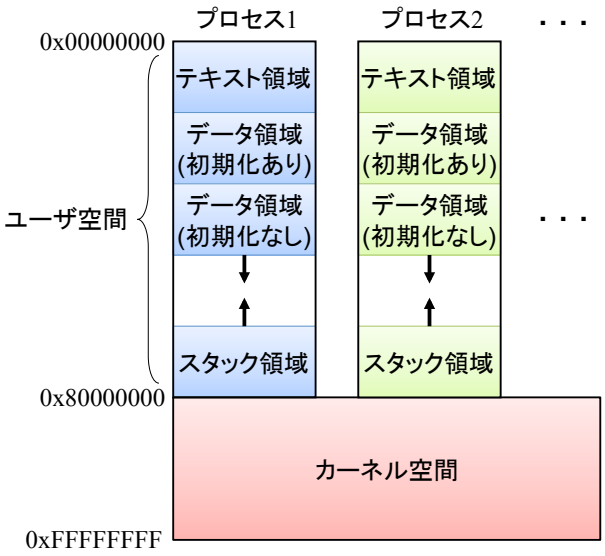

図 1.3: プロセスの仮想メモリ空間構造

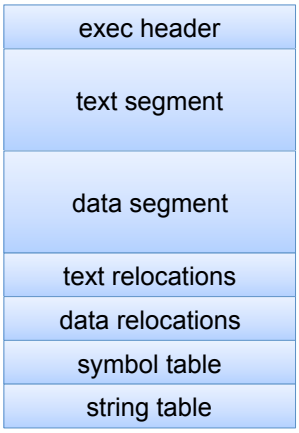

図 1.4: 実行形式ファイル (a.out) の構造

上が低位アドレス，下が高位アドレスを示す．たとえば ls プログラムが実行されると，ls ファイルのテキストセグメントが仮想メモリ空間のテキスト領域にロードされ，データセグメントがデータ領域にロードされる．さらにその上位に初期化されていない広域変数の領域が取られ，ユーザ空間の最上位の領域にはスタック領域が取られる．スタックの役割についてはプログラミング第 3 同演習で学習済みであるはずである．関数呼び出しなどによってスタック領域は低位番地に向かって伸びてゆく．一方，malloc() などによって動的にメモリ領域を割り当てることにより，データ領域は高位番地に向かって伸びてゆく．

ここで本題からはややそれるが，UNIX における実行ファイル形式を紹介する．現在は ELF (Executable and Linking Format) と呼ばれる形式が用いられているが，ここではより理解しやすいものとして古い形式である a.out 形式を紹介することとする．図 1.4 に a.out ファイルの構造を示す．

- exec header: 実行形のバイナリファイルをメモリにロードして実行する際のパラメータを格納．

- text segment: マシン命令 (機械語) を格納．メモリには通常 read-only としてロードされる．
- data segment: 初期化済みの広域変数を格納．メモリには通常 read-write 可能としてロードされる．
- text relocations: バイナリファイルを構成する際，text segment 内のポインターを更新するためにリンカーが使用した情報を格納する．
- data relocations: バイナリファイルを構成する際，data segment 内のポインターを更新するためにリンカーが使用した情報を格納．
- symbol table: シンボル (変数名や関数名) とアドレスの対応付けの情報を格納．
- string table: シンボルに対応した文字列を格納．

ユーザがプログラムの実行を開始すると，オペレーティングシステムは上記の内容にしたがってプロセスに図 1.3 に示したテキスト領域やデータ領域を割り当て，プロセスの実行を開始する．すなわち，a.out ファイル内の text segment の内容は仮想メモリ空間のテキスト領域にロードされ，a.out ファイルの data segment の内容は仮想メモリ空間のデータ領域 (初期化あり) にロードされる．また，a.out 内のパラメータから仮想メモリ空間のデータ領域 (初期化なし) のサイズが決められる．

1.5 ファイルによる入出力デバイスの抽象化

UNIX システムでは，データを入力したり出力したりする対象を**ファイル**として抽象化している．いわゆる通常のファイル (ディスク上に作られる情報の塊り) も “ファイル” であり，入出力デバイスも “ファイル” である．たとえば，キーボードは入力デバイスであり画面は出力デバイスであるが，これらも “ファイル” として扱うことができる．この他にも，ディスクそのものやネットワークインタフェースなどもデータ入出力のための “ファイル” として扱うことができる．

UNIX システムのコマンドは，**標準入力**と呼ばれるファイルから入力データを読み込み，**標準出力**と呼ばれるファイルに実行結果を出力するように作られているものが多い．実際はキーボードから入力データを読み込み，画面に実行結果が出力される．これは，プログラム開始時に標準入力がキーボードに結び付けられ，また標準出力が画面に結び付けられるからである．

UNIX コマンドの実行においては，**リダイレクト**によってたとえば実行結果をファイルにセーブするという使い方ができることは，2 年次のコンピュータ実習で学んでいると思う．リダイレクトとは，標準入力や標準出力をキーボードや画面とは別の “ファイル” に付け替えることである．詳しくは第 5.1.2 節で述べる．

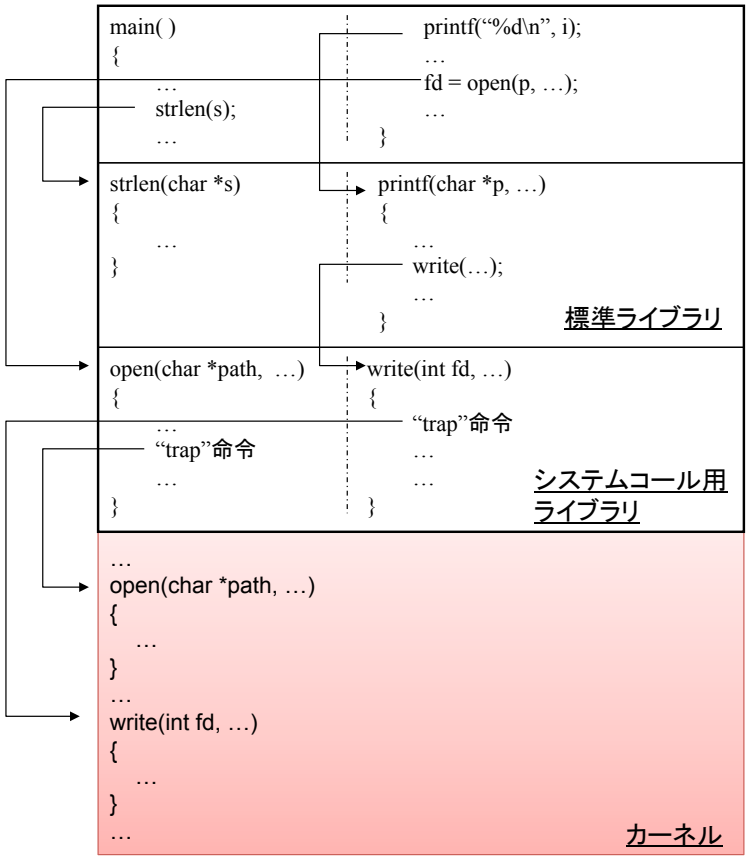

図 1.5: ライブラリとシステムコール

1.6 ライブラリとシステムコール

UNIX 上でプログラムを作成する際，さまざまな機能を実現する関数がライブラリやシステムコールとして用意されている．C 言語用の標準ライブラリは，多くのシステムでは `/usr/lib/libc.a` というファイルで提供されている．たとえば `strlen()` は文字列の長さを返すライブラリ関数であり，標準ライブラリに含まれている．ライブラリ関数はコンパイルの際，`a.out` ファイルにリンクされる．

一方，カーネルが提供する機能を直接呼び出すのがシステムコールである．たとえば，ファイル入出力ためにファイルをオープンするためには `open()` というシステムコールを用いる．システムコールもライブラリとして用意されているが，その中では単にカーネルを呼び出す機械語 (たとえば `trap` 命令) が実行されるだけであり，機能そのものはカーネル内で実現されている．

`strlen()` のようなライブラリ関数はその中だけで機能を実現しているが，たとえば `printf()` のようなライブラリ関数は標準出力への出力を伴うため，ライブラリ関数の中からシステムコールを呼び出している．以上を図に示すと，図 1.5 のようになる．

ライブラリ関数呼び出しもシステムコール呼び出しも C 言語上の記述方法は同じであるが，システムコールを利用する際にはいくつか注意すべきことがある．これらについての詳細は第 4.1 節で述べる．

1.7 man を使いこなそう

今後この科目ではライブラリ関数やシステムコールを利用することになる．これらの使い方や機能については man コマンドを利用しよう．ライブラリ関数やシステムコール以外にも，コマンドの使い方や制御データのフォーマットなども調べることができる．man コマンド自体の使い方も “man man” を実行すれば調べられる．man コマンドにもいろいろなスイッチが指定できるが，最も一般的な使い方は以下のとおりである．

man コマンドの使い方

```
% man [section] name ...
```

section は省略可能であるが，同じ name が複数のセクションに存在する場合は，明示的にセクションを指定する．たとえば write というコマンドもあり，システムコールもある．システムコールの write を調べたい場合は “man 2 write” と実行する．

セクションは番号で表され，以下のようになっている．

- 1: コマンド
- 2: システムコール
- 3: ライブラリ関数
- 4: カーネルインタフェース
- 5: ファイルフォーマット
- 6: ゲーム
- 7: misc.
- 8: システム管理者用

無闇にガイドブックや Web に頼ることなく，man コマンドを使いこなそう．

課題 1: Dijkstra の Shortest Path First アルゴリズムによる最小コスト経路計算

課題 1 のねらいは，明示されているアルゴリズムから C 言語によるプログラムを作成することである．本来は問題を解決するためのアルゴリズムは自分で考え出すものだが，小手調べとしてアルゴリズムは既知とし，プログラム作成に専念するものとする．

課題は，3 年春学期の「ネットワーク工学 I」で扱った，Dijkstra の Shortest Path First アルゴリムによる最小コスト経路の計算である．ネットワークは多数のルータやエンドノードおよびそれらを接続するリンクから構成されているが，これをモデル化し，ネットワークはノードとそれらを接続するリンクから構成されているとする．リンクにはそれぞれコストが割り当てられている．このとき，あるノードから他のすべてのノードへの最小コスト経路を求めたい．Dijkstra の Shortest Path First アルゴリズム (以下，SPF アルゴリズムと表記) はこのような最小コスト経路を求めるアルゴリズムの 1 つである．

Dijkstra の SPF アルゴリズムの概略

SPF アルゴリズムでは，ネットワーク内のすべてのノードがネットワークのトポロジーを知っていることが前提となる．ネットワークのトポロジーとは，ノードやリンクがどのように接続しているかということである．どのようにしてすべてのノードがネットワークのトポロジーを知ることができるかについては,「ネットワーク工学 I」を参照して欲しい．あるノードが SPF アルゴリズムを実行することにより，そのノードを根とし，各ノードまでの最小コスト経路を示す木構造を得ることができる．

図 1.6 に SPF アルゴリズムにおける記法を示す．また，図 1.7 に SPF アルゴリズムを示す．SPF アルゴリズムは初期化フェーズ (1〜6 行目) とループフェーズ (7〜12 行目) からなる．ループを 1 回実行することにより，ある 1 つのノードへの最小コスト経路が決定される．最初はすべてのノードは N に属しており，N' は空集合である．このとき，ノード u から他のすべてのノードへの最小コスト経路を求めるとする．初期化フェーズでは，まず自身 (u) を最小コスト経路が分かっているノードの集合である N' に加える (2 行目)．そして他のすべてのノード v について，u と v が隣接していれば u から v へのコスト $D(v)$ を u と v 間のリンクコスト $c(u,v)$ に設定し，v への最小コスト経路の 1 つ前のノードを u と設定する (5 行目)．もし，u と v が隣接していなければ u から v へのコスト $D(v)$ を無限大とする (6 行目)．

次にループフェーズに入る．まず N' に属していないノードの中から，u からのコストが最小であるノード w を見つけ，w を N' に加える (8 行目)．次に w の隣接ノードのうち N' に含まれていない v を見つけ (9 行目)，もし，現在のステップにおける u から v へのコスト $D(v)$ が，u から w を経由して v へ至る経路のコスト $(D(w)+c(w,v))$ よりも大きい場合は，$D(v)$ と $p(v)$ の値を更新する (11 行目)．以上の処理を，N' と N が等しくなるまで続ける．

- N: ノードの集合
- u: 始点ノード
- v: 終点ノード
- $c(i,j)$: ノード i からノード j へのリンクのコスト．i と j が隣接していない場合は無限大とする．
- $D(v)$: 始点 u から終点 v までの反復計算における現在のステップにおける最小コスト．
- $p(v)$: 始点 u から終点 v への現在のステップにおける最小コスト経路上における v の 1 つ手前のノード．
- N': 最小コスト経路が最終的に分かっているノードの集合．v までの最小コスト経路が分かっていれば，v は N' の要素．

図 1.6: Dijkstra の SPF アルゴリズムにおける記法

SPF アルゴリズムの実行例

図 1.8 に示すネットワークにおいて，ノード A で SPF アルゴリズムを実行したときの様子を図 1.9 に示す．Step 0 は初期化フェーズである．まず N' に自分自身，すなわち A を加える．そしてノード A と隣接関係にあるノード B, C, D には $D(v)$ の値としてリンクコストを設定し，$p(v)$ を A に設定する．隣接関係にないノード E, F については $D(v)$ を無限大に設定する．

続いてループフェーズに入る．Step 1 では，まず Step 0 において N' に含まれないノードの中から $D(v)$ が最小であるノードを見つける．この例ではノード D となる．(各ステップで選ばれたノードを下線で示している．) そこでノード D を N' に加える．続いて，ノード D と隣接し，かつ N' に含まれないノード (この例ではノード B, C, E) について $D(v)$ を再計算する．ノード B については，A→D→B のコストは 3 となり，前ステップでのコストが 2 であるので，$D(B)$ は変化しない．ノード C については，A→D→C のコストは 4 となり，前ステップでのコストである 5 よりも小さいので，$D(C)$ を 5 から 4 に変更し，$p(C)$ も A から D へ変更する．ノード E については，A→D→E のコストは 2 となり，前ステップでのコストが無限大であるので，$D(E)$ を 2 に設定し，$p(E)$ を D に設定する．これで Step 1 が終わる．

続く Step 2 では，まず Step 1 での $D(v)$ が最小であるノードを見つける．この例ではノード B とノード E がコスト 2 であるが，ここではノード E を選択することとする．そこで E を N' に加える．続いて，ノード E と隣接し，かつ N' に含まれないノード (この例ではノード C, F) について $D(v)$ を再計算する．ノード C については，A→D→E→C のコストは 3 となり，前ステップでのコストである 4 より小さいので，$D(C)$ を 4 から 3 に変更し，$p(C)$ も D から E に変更する．ノード F については，A→D→E→F のコストは 4 と

```
1   Initialization
2       N' = u
3       for all nodes v
4           if v is a neighbor of u
5               then D(v) = c(u,v), p(v) = u
6           else D(v) = ∞
7   Loop
8       find w not in N' such that D(w) is a minimum, and add w to N'
9       update D(v) (and p(v)) for each neighbor v of w and not in N':
10          if D(v) > D(w) + c(w,v)
11              then D(v) = D(w) + c(w,v), p(v) = w
12  until N' = N
```

図 1.7: Dijkstra の Shortest Path First アルゴリズム

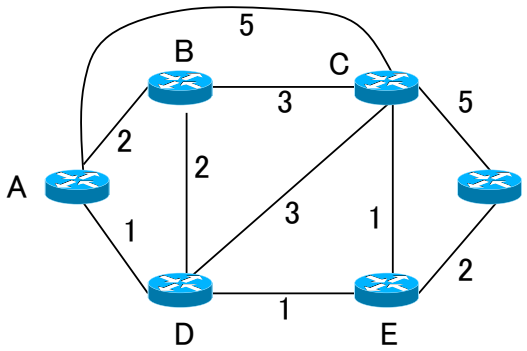

図 1.8: ネットワークトポロジーの例

なり，前ステップでのコストが無限大であるので，$D(F)$ を 4 に設定し，$p(F)$ を E に設定する．ノード B については再計算する必要はないので，前ステップの結果をそのまま使用する．以下，Step 3〜5 も同様に行う．

この結果，各ステップにおいて N' に加えたノード (下線を引いたノード) を見ていくと，次の結果を得ることができる．

- Step 1: $D(D) = 1, p(D) = A$ (D への最小コストは 1 で，1 つ前のノードは A)
- Step 2: $D(E) = 2, p(E) = D$ (E への最小コストは 2 で，1 つ前のノードは D)
- Step 3: $D(B) = 2, p(D) = A$ (B への最小コストは 2 で，1 つ前のノードは A)
- Step 4: $D(C) = 3, p(D) = E$ (C への最小コストは 3 で，1 つ前のノードは E)
- Step 5: $D(F) = 4, p(D) = E$ (F への最小コストは 4 で，1 つ前のノードは E)

上記から図 1.10 に示す木構造を描くことができる．

Step	N'	D(B),p(B)	D(C),p(C)	D(D),p(D)	D(E),p(E)	D(F),P(F)
0	A	2,A	5,A	1,A	∞	∞
1	AD	2,A	4,D		2,D	∞
2	ADE	2,A	3,E			4,E
3	ADEB		3,E			4,E
4	ADEBC					4,E
5	ADEBCF					

図 1.9: ノード A での実行例

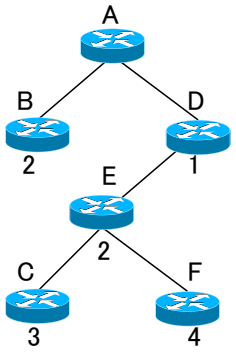

図 1.10: ノード A を根とした最小コスト経路の木構造

課題の説明

課題は図 1.8 に示したネットワークにおいて，各ノードを始点とする最小コスト経路を計算するプログラムを作成することである．今回は入力データとなるネットワークのトポロジーが決まっているので，プログラム中に，図 1.11 に示すような入力データを埋め込んでしまうこととする．`NNODE` はネットワーク内のノード数 (6) を表す文字定数である．`INF` は無限大を表す文字定数である．`int linkcost[][]` はノード間のリンクのコストを表す 2 次元配列である．

計算結果を格納する変数として，たとえば各ノードへの最小コストを格納するために `int mincost[NNODE]` を定義し，また各ノードへの最小コスト経路における 1 つ手前のノードを格納するために `int prev[NNODE]` を定義する．そして図 1.12 のように結果を出力する．この結果の表示において，“`root node:  A`” は次の行にノード A を始点として各ノードへの最小コスト経路を計算した結果であることを示している．たとえば 2 行目の右端にある `[F, E, 4]` は，ノード A からノード F への最小コスト経路の 1 つ手前はノード E であり，最小コストは 4 であることを示している．

```
#define NNODE    6        // number of nodes
#define INF      100      // infinity

// link cost matrix
int linkcost[NNODE][NNODE] = {
  {  0,   2, 5,   1, INF, INF},  // from node-A to other nodes
  {  2,   0, 3,   2, INF, INF},  // from node-B to other nodes
  {  5,   3, 0,   3,   1,   5},  // from node-C to other nodes
  {  1,   2, 3,   0,   1, INF},  // from node-D to other nodes
  {INF, INF, 1,   1,   0,   2},  // from node-E to other nodes
  {INF, INF, 5, INF,   2,   0}   // from node-F to other nodes
};
```

図 1.11: プログラム中で定義する入力データ例

```
root node: A
  [A, A, 0] [B, A, 2] [C, E, 3] [D, A, 1] [E, D, 2] [F, E, 4]
root node: B
  . . .
. . .
root node: F
  . . .
```

図 1.12: 出力形式