8. Linux のシステムプログラミングに関する知識Ⅱ

1. 科目の概要

Linux においてシステムが提供するリソースを活用してプログラムを実行するシステムプログラミングのうち、スレッドプログラミングやプロセス間通信、デバイスファイルの利用、セマフォや共有メモリ、メッセージキューの利用など、比較的高度な話題について解説する。

2. 習得ポイント

本科目の学習により習得することが期待されるポイントは以下の通り。

習得ポイント	説　明	シラバスの対応コマ
II-8-1. 複数ソースプログラムのビルド	多くのソフトウェアでは、ひとつのソースコードで全てのコードが記述されるわけではない。ソースコードを分割することのメリットについて説明し、複数のソースコードからアプリケーションをビルドするためのツールであるmakeの概念と利用方法について解説する。	8
II-8-2. ソースコード管理と差分情報	複数ファイルから構成されるプログラムを多人数で開発する環境においては、ソースコードをまとめて管理する必要がある。ソースコード管理の考え方を示し、ソースコード管理ツールの代表的なツールであるRCSやSubversionなどを紹介する。またこれらのツールにおける基本的な概念である差分管理と、差分情報を取り扱うツールについて説明する。	8
II-8-3. デバッグの基本	C言語プログラミングにおけるデバッグの方法を概説する。デバッグプリントの挿入、デバッグ情報の埋め込み方に始まり、デバッガの利用方法、デバッグに利用するシンボル情報を有効にするコンパイル、ブレークポイントやステップ実行といったデバッグの基本的な作業について説明する。	9
II-8-4. プロセスの生成と管理	プロセスの概念を簡単に説明し、forkシステムコールによって新たにプロセスを生成する方法、プロセス管理情報を得る方法、procファイルシステムを利用したプロセス情報の取得など、プロセスの生成と管理を行うプログラミング手法を紹介する。	10
II-8-5. スレッドプログラミング	複数プロセスの利用よりも軽量な並列プログラムを実現するスレッドプログラミングについて解説する。プロセスとスレッドの違いについて述べ、pthreadライブラリを利用してスレッドプログラミングを実現する方法を説明する。	10
II-8-6. シグナルの利用	シグナルを利用して非同期プログラミングを実現する方法を説明する。シグナルの種類やシグナルの取り扱い方法、タイマー割り込みプログラムの作成方法、プログラムの強制停止とシグナルハンドラ等の概念について述べる。	11
II-8-7. パイプによるプロセス間通信	プロセス間通信を実現する手法のひとつとして、パイプを利用した通信手順を紹介する。パイプの作成と入出力制御、名前付きパイプの利用方法などについて、関連する関数を利用してプロセス間通信を実現するための具体的な手順を説明する。	12
II-8-8. デバイスファイルの扱い方	キーボード、ディスプレイ、プリンタ、ディスクといった全てのデバイスをファイルとして一元的に取り扱う考え方を説明する。また、入出力先を変更するリダイレクトについて述べ、デバイスファイルを簡単に利用する方法を紹介する。	13
II-8-9. セマフォ、共有メモリ、メッセージキューの利用	プロセス間通信で必要な手段であるセマフォ、共有メモリ、メッセージキューについてそれぞれの概念を理解させる。セマフォを利用した排他制御、共有メモリを利用したプロセス間のデータ転送、メッセージキューを利用したプロセス間通信の具体的な方法を解説する。	14
II-8-10. ソケットによるネットワーク通信	ネットワークを超えた通信を可能とするソケットの概念を解説する。さらにソケットを用いたネットワーク通信の具体的な手順を示し、サーバプログラムの簡単な動作やホストバイトオーダとネットワークバイトオーダの違いといった関連する話題についても触れる。	15

【学習ガイダンスの使い方】

1. 「習得ポイント」により、当該科目で習得することが期待される概念・知識の全体像を把握する。
2. 「シラバス」、「IT 知識体系との対応関係」、「OSS モデルカリキュラム固有知識」をもとに、必要に応じて、従来の IT 教育プログラム等との相違を把握した上で、具体的な講義計画を考案する。
3. 習得ポイント毎の「学習の要点」と「解説」を参考にして、講義で使用する教材等を準備する。

3. IT 知識体系との対応関係

「8. Linux のシステムプログラミングに関する知識Ⅱ」と IT 知識体系との対応関係は以下の通り。

科目名	基本レベル(Ⅰ)							応用レベル(Ⅱ)							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8. Linuxのシステムプログラミングに関するスキル	<ログイン手順とコンパイル手順>	<shellプログラミング>	<ファイル入出力プログラミング>	<ファイルシステム>	<UNIX環境>	<ライブラリの利用方法と作成手順>	<データの管理>	<ソフトウェアの開発環境>	<デバッグ>	<プロセスとスレッド>	<シグナル>	<プロセス間通信とパイプ>	<端末機器の入出力>	<セマフォ、共有メモリ、メッセージキュー>	<ネットワークプログラミング>

[シラバス：http://www.ipa.go.jp/software/open/ossc/download/Model_Curriculum_05_08.pdf]

<IT 知識体系上の関連部分>

分野		科目名	1	2	3	4	5	6	7	8	9	10	11	12	13
組織関連事項と情報システム	1	IT-IAS 情報保証と情報セキュリティ	IT-IAS1. 基礎的な問題	IT-IAS2. 情報セキュリティの仕組み(対策)	IT-IAS3. 運用上の問題	IT-IAS4. ポリシー	IT-IAS5. 攻撃	IT-IAS6. 情報セキュリティ分野	IT-IAS7. フォレンジック(情報証拠)	IT-IAS8. 情報の状態	IT-IAS9. 情報セキュリティサービス	IT-IAS10. 脅威分析モデル	IT-IAS11. 脆弱性		
	2	IT-SP 社会的な観点とプロフェッショナルとしての課題	IT-SP1. プロフェッショナルとしてのコミュニケーション	IT-SP2. コンピュータの歴史	IT-SP3. コンピュータを取り巻く社会環境	IT-SP4. チームワーク	IT-SP5. 知的財産権	IT-SP6. コンピュータの法的問題	IT-SP7. 組織の中のIT	IT-SP8. プロフェッショナルとしての倫理的な問題と責任	IT-SP9. プライバシーと個人の自由				
応用技術	3	IT-IM 情報管理	IT-IM1. 情報管理の概念と基礎	IT-IM2. データベース問合わせ言語	IT-IM3. データアーキテクチャ	IT-IM4. データモデリングとデータベース設計	IT-IM5. データと情報の管理	IT-IM6. データベースの応用分野							
	4	IT-WS Webシステムとその技術	IT-WS1. Web技術	IT-WS2. 情報アーキテクチャ	IT-WS3. デジタルメディア	IT-WS4. Web開発	IT-WS5. 脆弱性	IT-WS6. ソーシャルソフトウェア							
ソフトウェアの方法と技術	5	IT-PF プログラミング基礎	IT-PF1. 基本データ構造	IT-PF2. プログラミングの基本的構成要素	IT-PF3. オブジェクト指向プログラミング	IT-PF4. アルゴリズムと問題解決	IT-PF5. イベント駆動プログラミング	IT-PF6. 再帰							
	6	IT-IPT 技術を統合するためのプログラミング	IT-IPT1. システム間通信	IT-IPT2. データ割り当てと変換	IT-IPT3. 統合的コーディング	IT-IPT4. スクリプティング手法	IT-IPT5. ソフトウェアセキュリティの実現	IT-IPT6. 種々の問題 [8-8]	IT-IPT7. プログラミング言語の概要						
	7	CE-SWE ソフトウェア工学	CE-SWE0. 歴史と概要	CE-SWE1. ソフトウェアプロセス	CE-SWE2. ソフトウェアの要求と仕様	CE-SWE3. ソフトウェアの設計	CE-SWE4. ソフトウェアのテストと検証	CE-SWE5. ソフトウェアの保守	CE-SWE6. ソフトウェア開発・保守ツールと環境 [8-8]	CE-SWE7. ソフトウェアプロジェクト管理	CE-SWE8. 言語翻訳 [8-8]	CE-SWE9. ソフトウェアのフォールトトレランス	CE-SWE10. ソフトウェアの構成管理	CE-SWE11. ソフトウェアの標準化	
	8	IT-SIA システムインテグレーションとアーキテクチャ	IT-SIA1. 要求仕様	IT-SIA2. 調達/手配	IT-SIA3. インテグレーション	IT-SIA4. プロジェクト管理	IT-SIA5. テストと品質保証	IT-SIA6. 組織の特性	IT-SIA7. アーキテクチャ						
システム基盤	9	IT-NET ネットワーク	IT-NET1. ネットワークの基礎	IT-NET2. ルーティングとスイッチング	IT-NET3. 物理層	IT-NET4. セキュリティ	IT-NET5. アプリケーション分野	IT-NET6. ネットワーク管理							
	10	CE-NWK テレコミュニケーション	CE-NWK0. 歴史と概要	CE-NWK1. 通信ネットワークのアーキテクチャ	CE-NWK2. 通信ネットワークのプロトコル	CE-NWK3. LANとWAN	CE-NWK4. クライアントサーバコンピューティング	CE-NWK5. データのセキュリティと整合性	CE-NWK6. ワイヤレスコンピューティングとモバイルコンピューティング	CE-NWK7. データ通信	CE-NWK8. 組込み機器向けネットワーク	CE-NWK9. 通信技術とネットワーク概要	CE-NWK10. 性能評価	CE-NWK11. ネットワーク管理	CE-NWK12. 圧縮と伸張
			CE-NWK13. クラスタシステム	CE-NWK14. インターネットアプリケーション	CE-NWK15. 次世代インターネット	CE-NWK16. 放送									
	11	IT-PT プラットフォーム技術	IT-PT1. オペレーティングシステム	IT-PT2. アーキテクチャと機構	IT-PT3. コンピュータインフラストラクチャ	IT-PT4. デプロイメントソフトウェア	IT-PT5. ファームウェア	IT-PT6. ハードウェア							
	12	CE-OPS オペレーティングシステム	CE-OPS0. 歴史と概要 [8-4]	CE-OPS1. 並行性	CE-OPS2. スケジューリングとディスパッチ	CE-OPS3. メモリ管理 [8-7]	CE-OPS4. セキュリティと保護	CE-OPS5. ファイル管理	CE-OPS6. リアルタイムOS	CE-OPS7. OSの概要	CE-OPS8. 設計の原則	CE-OPS9. デバイス管理	CE-OPS10. システム性能評価		
コンピュータアーキテクチャ	13	CE-CAO コンピュータのアーキテクチャと構成	CE-CA00. 歴史と概要	CE-CA01. コンピュータアーキテクチャの基礎	CE-CA02. メモリシステムの構成とアーキテクチャ	CE-CA03. インタフェースと通信	CE-CA04. デバイスサブシステム	CE-CA05. CPUアーキテクチャ	CE-CA06. 性能・コスト評価	CE-CA07. 分散・並列処理	CE-CA08. コンピュータによる計算	CE-CA09. 性能向上			
複数領域にまたがるもの	14	IT-ITF IT基礎	IT-ITF1. ITの一般的なテーマ	IT-ITF2. 組織の問題	IT-ITF3. ITの歴史	IT-ITF4. IT分野(学科)とそれに関連のある分野(学科)	IT-ITF5. 応用領域	IT-ITF6. IT分野における数学と統計学の活用							
	15	CE-ESY 組込みシステム	CE-ESY0. 歴史と概要	CE-ESY1. 低電力コンピューティング	CE-ESY2. 高信頼性システムの設計	CE-ESY3. 組込み用アーキテクチャ	CE-ESY4. 開発環境	CE-ESY5. ライフサイクル	CE-ESY6. 要件分析	CE-ESY7. 仕様定義	CE-ESY8. 構造設計	CE-ESY9. テスト	CE-ESY10. プロジェクト管理	CE-ESY11. 並行設計(ハードウェア、ソフトウェア	CE-ESY12. 実装
			CE-ESY13. リアルタイムシステム設計	CE-ESY14. 組込みマイクロコントローラ	CE-ESY15. 組込みプログラム	CE-ESY16. 設計手法	CE-ESY17. ツールによるサポート	CE-ESY18. ネットワーク型組込みシステム	CE-ESY19. インタフェースシステムと混合信号システム	CE-ESY20. センサ技術	CE-ESY21. デバイスドライバ	CE-ESY22. メンテナンス	CE-ESY23. 専門システム	CE-ESY24. 信頼性とフォールトトレランス	

4. OSS モデルカリキュラム固有の知識

OSS モデルカリキュラム固有の知識として、Linux という具体的な環境におけるプログラミング手法の知識が含まれる。デバッグ手法、プロセス管理、入出力管理などに関するプログラミング手法について、Linux を通して習得する。

科目名	第8回	第9回	第10回	第11回	第12回	第13回	第14回	第15回
8.Linux のシステムプログラミングに関する知識Ⅱ	(1)make と Makefile	(1)ソースプログラムにデバッグ情報を埋め込む方法	(1)プロセス	(1)シグナルの種類	(1)低水準の pipe 関数	(1)標準入出力、標準エラー出力をファイルにリダイレクトする手順を理解する。	(1)セマフォ（排他制御）	(1)ソケット
	(2)ソースコード管理	(2)gdb によるデバッグ作業	(2)プロセス管理情報	(2) SIGALRM とalarm を使用してタイマー割り込みプログラムを作成す	(2)高水準の popen と pclose 関数	(2)/dev ファイルを使った入出力処理。	(2)共有メモリ（Shared Memory）	(2)複数のクライアント
	(3)patch とtar		(3)/proc/数字ディレクトリ	(3)SIGINT を指定したプログラムを作成し、Ctrl-C キーを押した時の動作を確	(3)名前付きパイプ（FIFO）		(3)メッセージキュー	(3)現在サービスを受け付けているポート番号の確認
			(4)スレッド					

（網掛け部分は IT 知識体系で学習できる知識を示し、それ以外は OSS モデルカリキュラム固有の知識を示している）

スキル区分	OSS モデルカリキュラムの科目	レベル
システム分野	8　Linux のシステムプログラミングに関する知識 II II	応用
習得ポイント	II -8-1. 複数ソースプログラムのビルド	
対応するコースウェア	第 8 回 ソフトウェアの開発環境	

II -8-1. 複数ソースプログラムのビルド

多くのソフトウェアでは、ひとつのソースコードで全てのコードが記述されるわけではない。ソースコードを分割することのメリットについて説明し、複数のソースコードからアプリケーションをビルドするためのツールである make の概念と利用方法について解説する。

【学習の要点】

* プログラムのソースコードはしばしば複数のソースファイル（または他のファイル）から生成される。
* プログラムは、自分以外の人が書いたライブラリを組み込んで構成することがある。OS によって提供される機能（ライブラリやシステムコール）を利用する場合もこれに相当する。このような場合、ライブラリやシステムコールは、外部のプログラムから利用できるようにヘッダファイルと呼ばれるインタフェースを公開する。
* 外部にインタフェースを公開する以外にも、ソースコードの可読性を向上させたり、モジュール性を持たせたりという目的のために分割することがある。
* make コマンドは、対象ソースファイル、参照ヘッダファイル、参照ライブラリ、およびコンパイル・リンクオプションの書かれたファイル（makefile）を読み込んで、プログラムのビルドを自動化するツールである。

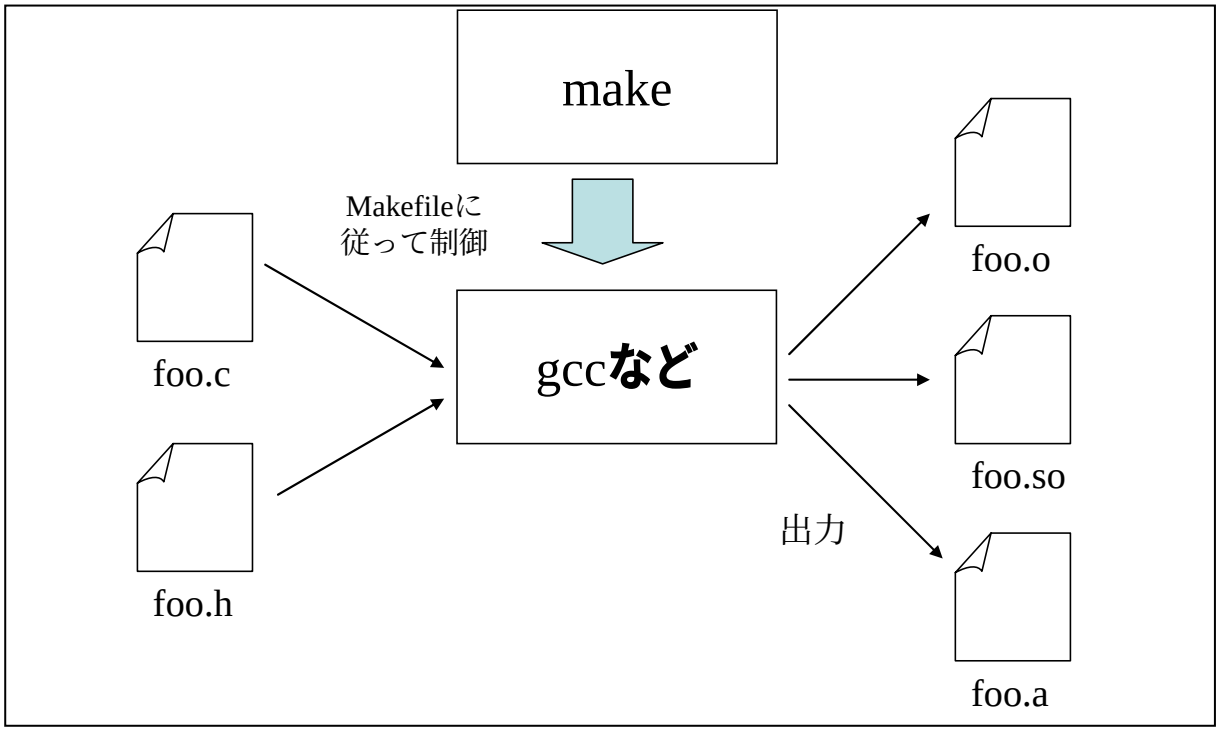

図 II-8-1.　make の流れ

【解説】

1) ソースコードを分割することのメリットとデメリット

* メリット
  - ヘッダファイルと呼ばれるインタフェースを使って、ライブラリとして外部に公開したとき、汎用性が高まる。
  - 個々のモジュールを小さくすることにより、モジュール性を高めることができる。
  - デバッグ時に問題の切り分けがより単純になる。また、ソースコードの可読性が向上する。
  - 一度ソースファイルをコンパイルすれば、変更を加えない限り再コンパイルする必要がなくなるので、プログラムをコンパイルする合計時間を短縮できる。
* デメリット
  - 過度に細かく分割した場合、ソースファイル数が膨れ上がり、可読性は逆に低下してしまう。

2) ヘッダファイル

* 関数プロトタイプ宣言が記述されたファイルのこと。ライブラリのインタフェースとして機能する。
* そのライブラリを使用するソースファイルをコンパイルするとき、ヘッダファイルを使って、関数の呼び出しが正しいかチェックする。

3) make とは

* 複数のソースファイルからなるプログラムのビルドを、自動化するためのツールのこと。
* C や C++だけでなく、TeX やデータベース管理などにも適用できる。
* メリット
  - 不要な再コンパイルを避ける事ができ、ビルド時間を短縮できる。
  - コンパイルだけでなく、モジュールやライブラリとのリンク方法も定義できる。
  - プログラム自体の複雑さが増大するほど、コンパイルやリンクを手作業で行う際に発生するミスが増える。make によりそれらを避けることができる。
* make を使用するためには、プログラムをビルドするためのルールが記されているファイル（ビルドファイル)「makefile」を作成する必要がある。
  - 最初のルールは基本的に、ターゲット、必須項目、実行コマンドから構成される。
  - それらのルールに、対象ソースファイル、参照ヘッダファイル、参照ライブラリ、およびコンパイル・リンクオプションなどが記述される。
  - 書かれた要素間の依存関係をもとに、make はファイルのタイムスタンプの比較を行うなどして、不必要なコンパイル作業を除去し、ビルドを最適化する。
* Java においては make の代わりに作られたビルドツールとして Ant がある。Ant も makefile と同様にビルドファイルを持つが、それは XML により記述される。

スキル区分	OSS モデルカリキュラムの科目	レベル
システム分野	8　Linux のシステムプログラミングに関する知識 II II	応用
習得ポイント	II-8-2. ソースコード管理と差分情報	
対応するコースウェア	第 8 回 ソフトウェアの開発環境	

II-8-2. ソースコード管理と差分情報

複数ファイルから構成されるプログラムを多人数で開発する環境においては、ソースコードをまとめて管理する必要がある。ソースコード管理の考え方を示し、ソースコード管理ツールの代表的なツールである RCS や Subversion などを紹介する。

【学習の要点】

* プログラムを構成するファイルを管理するということは、それを構成するファイル一覧とその世代を管理することである。
* 複数人でプログラムを開発するときは、互いに変更点が競合するのを防ぐ必要がある。これには、人的なコミュニケーションが不可欠であるが、これを補助するツールが存在する。RCS や Subversion はその一例である。
* RCS や Subversion といったバージョン管理ツールは、関連するソースコード一覧にタグをつけて範囲を管理する。その上で、範囲内の個々のソースコードの世代間差分を保持・復元することでプロジェクトの管理を行うものである。

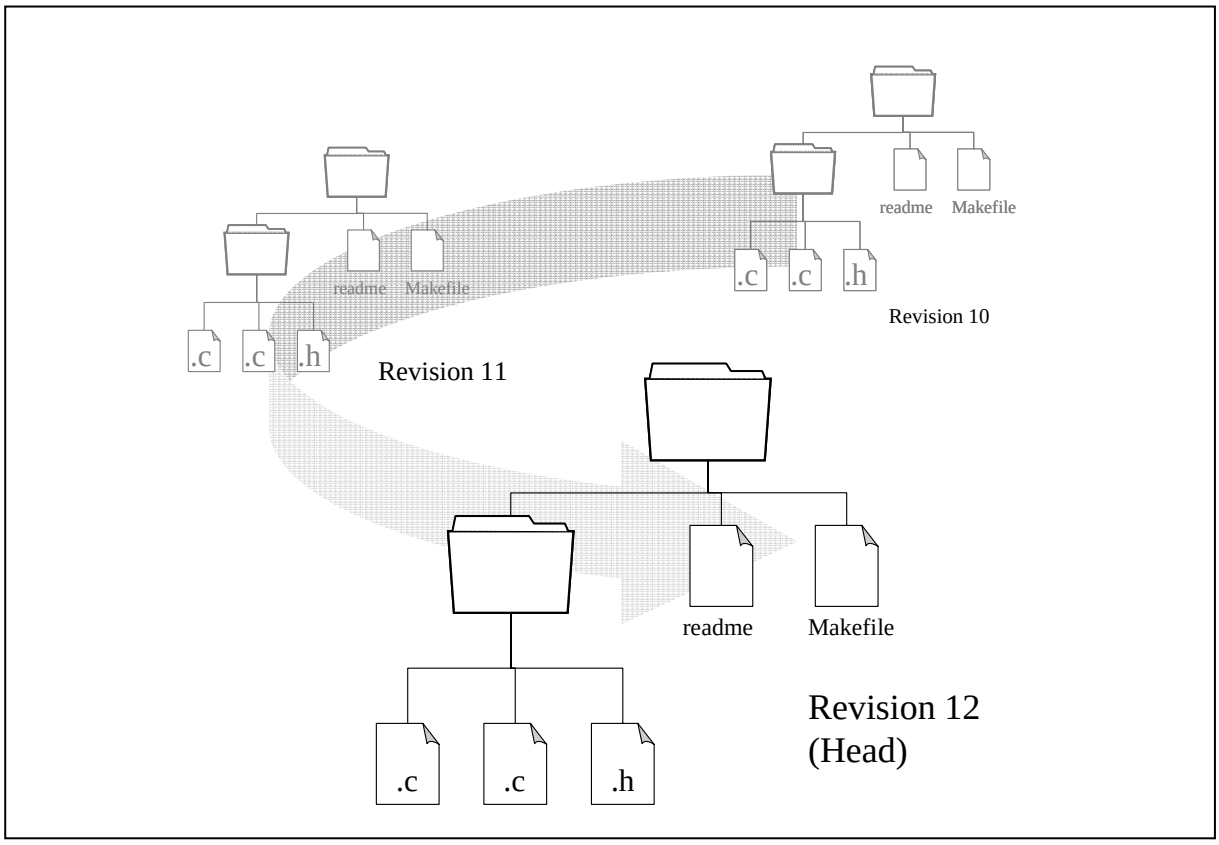

図 II-8-2.　プロジェクト管理

【解説】

1) バージョン管理

* 主にテキストファイルのバージョン（リビジョン）を管理し、必要に応じて過去のバージョンのファイルを取得できるような機構をバージョン管理機能という。
* 有名なバージョン管理システムにRCS (Revision Control System) がある。RCSは、ファイルの新旧を、差分（前のバージョンとの変更点情報）で表す。すべてのバージョンの生のファイルを保持するわけではない。
* RCS は、古いファイルを得る要求を受けると、ヘッド（最新）バージョンから所望のバージョンまで差分情報を元にファイルのコンテンツを計算する。

2) ファイル群の管理

* RCS は、ただひとつのファイルを管理するのであって、ファイル群に対してバージョン管理を行うことはできない。
* リポジトリ (バージョン管理するファイル群の保管場所) は、時間とともに伸縮する（ファイルは増えたり減ったりする）。また同時に、ファイルの内容は時間とともに変化する。近年のバージョン管理システムは、この 2 つの軸からなる空間から、ファイル群を選択できる能力が求められる。
* CVS は、ファイル群をモジュールとして管理することができ、ファイル一つ一つに独立してリビジョンを振っていく。
* Subversion は、ファイル群をディレクトリにて管理する。CVS と違って、Subversion はリポジトリに対してリビジョン番号が振られる。このため、Subversion では、ある時点のリポジトリ全体のスナップショットに容易にアクセスすることができる。

3) CVS でのプロジェクトの管理

* CVS では、リビジョンはファイル毎に付加されるため、ファイル群に対してバージョン管理を行なうには以下のように行うとよい。
  - モジュールを定義する
  - ある時点のモジュールに対してタグ（スナップショット名）を付ける
* ある時点のファイル群を取り出すには、モジュール名とモジュールのタグを指定する。
* タグを利用しない場合は、モジュール名と日付を指定する。

4) Subversion でのプロジェクトの管理

* Subversion では、リポジトリ毎にリビジョンが振られる。リポジトリに対してリビジョンを指定すると、そのリビジョン時点での全ファイルを選択できる。
* ある時点が決まれば、あとはファイル群を選択するだけである。Subversion ではファイル群はディレクトリで管理するので、ディレクトリ名を指定するだけでよい。
* CVS と同様、タグの概念を導入することもできる。Subversion では、タグもその名前のついたディレクトリにすぎない。

スキル区分	OSS モデルカリキュラムの科目	レベル
システム分野	8　Linux のシステムプログラミングに関する知識 II II	応用
習得ポイント	II-8-3. デバッグの基本	
対応する コースウェア	第 9 回 デバッグ	

II-8-3. デバッグの基本

C 言語プログラミングにおけるデバッグの方法を概説する。デバッグプリントの挿入、デバッグ情報の埋め込み方に始まり、デバッガの利用方法、デバッグに利用するシンボル情報を有効にするコンパイル、ブレークポイントやステップ実行といったデバッグの基本的な作業について説明する。

【学習の要点】

* プログラムの入出力をプログラマが検証するには、いくつかの方法がある。単純に画面に変数を出力する方法や、デバッガを使ってプログラムの実行を途中で止め、その後はインタラクティブに実行する方法などが用いられる。
* 通常リリース時には、デバッグのための情報を保持している必要はない。gcc のデフォルトの動きでは、デバッグ情報は保持されないので、gdb でデバッグする場合には、コンパイル時に明示的にデバッグ情報を保持するオプションをつける必要がある。
* C 言語において、gdb でプログラムの動作を途中で止めるには、止めたい部分の関数名を指定してブレークポイントを設定する。ブレークポイントに遭遇すると、その関数を実行する直前で処理が止まり、インタラクティブなモードに移る。

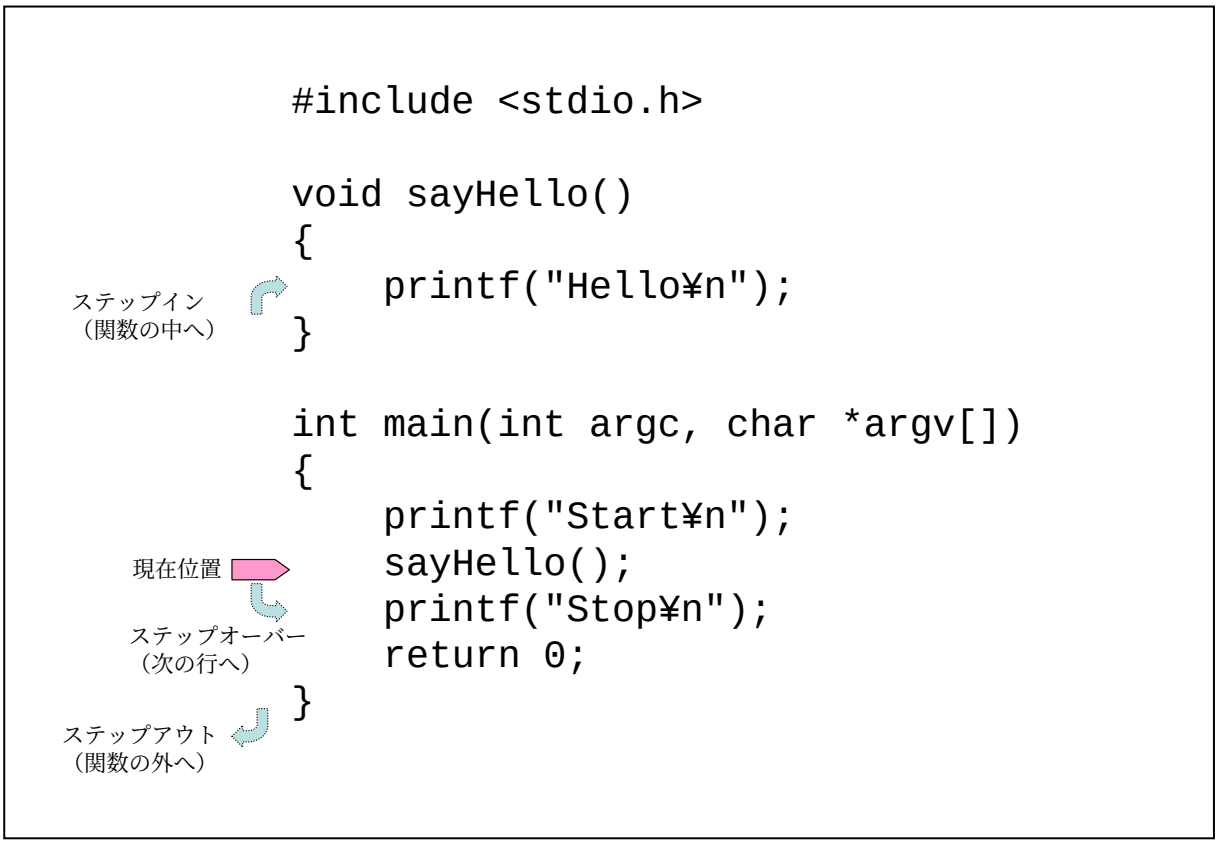

図 II-8-3.　ステップ実行

【解説】

1) デバッグのパターン

* 具体的なデバッグの作業は、プログラムのある時点で変数にどのような値がセットされているかどうか（メモリの状態）を見て、その値が正しいかどうか確認することと、期待する値と食い違いが生じたソースコード上の場所を発見することである。
* デバッグの方法にはいくつかのパターンがある。
  - 変数出力命令（デバッグプリント）の埋め込み
    最も原始的な方法で、ソースコード内に printf のような変数の内容を出力する命令を組み込み、実行時に出力される内容を確認する方法である。
  - デバッグシンボルの埋め込み
    コンパイル時にデバッグシンボルを残す（gcc では-g オプションをつける）方法。gdb のようなデバッガにてインタラクティブにプログラムをステップ実行し、必要であれば変数の内容やアドレスを出力することができる。

2) ブレークポイント

* デバッガでプログラムを実行する際に、プログラムを途中で止めるよう指定するのがブレークポイントである。gdb では、ブレークポイントとして、ソースコードの行番号や関数名などが指定できる。
* 統合開発環境などでは、視覚的に行にブレークポイントを設定することができる。しばしば、画鋲（ピン）のアイコンで表される。
* ブレークポイントでプログラムを止めた後は、ステップ実行（一行ずつ実行するなど）を行なって、不良箇所を発見する。

3) ステップ実行

* ブレークポイントでプログラムを止めたら、以下のような操作により、インタラクティブに実行を継続する。
  - ステップイン
    現在の位置がサブルーチンの呼び出しポイントである場合、サブルーチンの内部へ移動する。
  - ステップアウト
    現在のサブルーチンの残りの部分を実行し、その呼び出し元へ戻る。
  - ステップオーバー
    現在の位置から、ソースコードの次の行へ移動する。現在の行がサブルーチンの呼び出しポイントである場合は、そのサブルーチンの実行後の状態となる。
* 不正なメモリ参照を行なった場合などは、ステップ実行は途中で失敗する。失敗が起こると不良箇所は見つけ易い。ステップ実行により実行した部分に不良が存在することが多いからである。
* 論理的な不良をデバッガによって発見したい場合は、ステップ実行によりその都度メモリの状態を検証する必要がある。

スキル区分	OSS モデルカリキュラムの科目	レベル
システム分野	8　Linux のシステムプログラミングに関する知識 II II	応用
習得ポイント	II-8-4. プロセスの生成と管理	
対応する コースウェア	第 10 回 プロセスとスレッド	

II-8-4. プロセスの生成と管理

プロセスの概念を簡単に説明し、fork システムコールによって新たにプロセスを生成する方法、プロセス管理情報を得る方法、proc ファイルシステムを利用したプロセス情報の取得など、プロセスの生成と管理を行うプログラミング手法を紹介する。

【学習の要点】

* プロセスとは、カーネルによって管理される実行の単位である。すべてのプロセスはカーネルによってプロセス ID というユニークな番号が与えられる。
* あるプロセスが、新しくプロセスを生成するには、fork システムコールを用いる。fork によって生成されたプロセスは、それを呼び出したプロセスの子プロセスとなる。
* カーネルは proc インタフェースを使用してプロセス情報をユーザに公開する。通常/proc にマウントされる疑似ファイルシステムにあるプロセス ID が名前となったディレクトリ下のファイルから、そのプロセスに関する情報が取得できる。

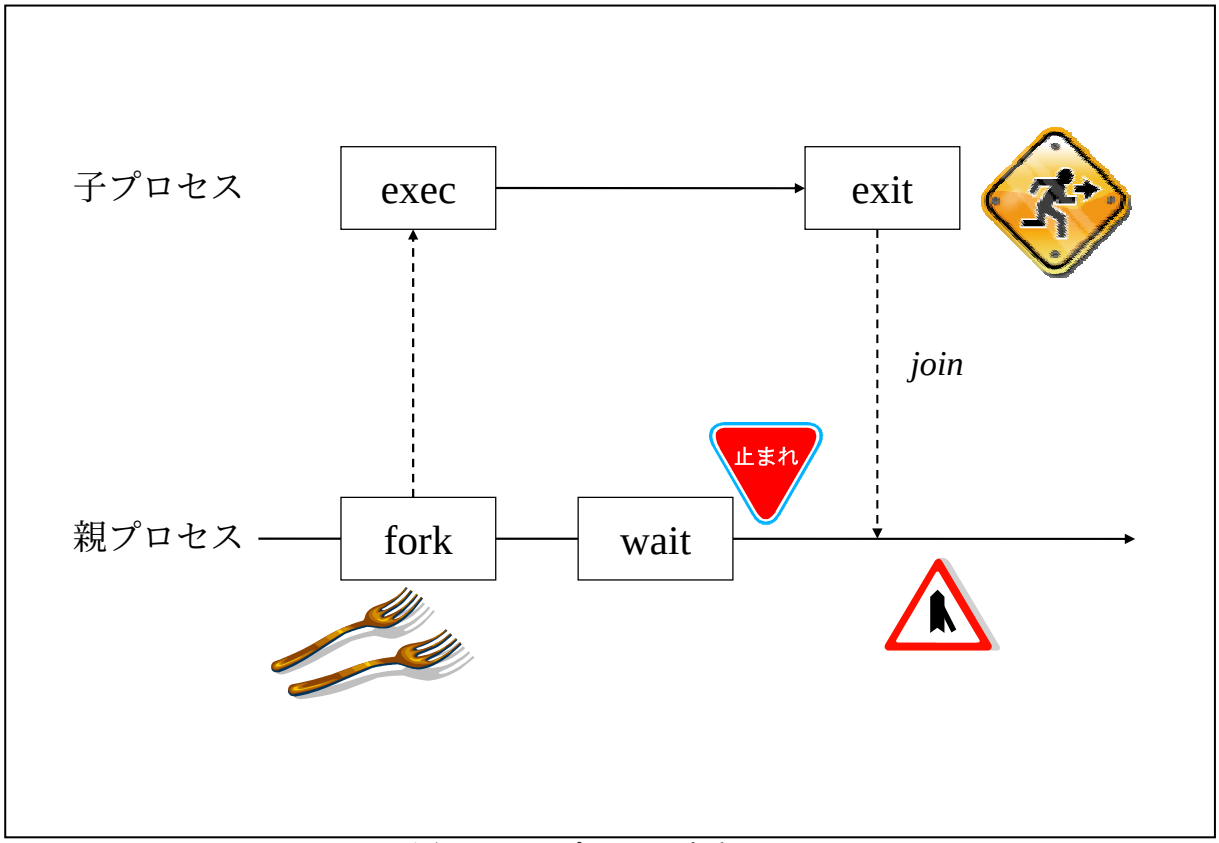

図 II-8-4.　プロセスの fork と join

【解説】

1) プロセスの生成

* あるユーザプロセスは、fork システムコールを使って自分自身の複製を生成することができる。複製によって生成されたプロセスは、fork を呼び出したプロセスの子プロセスと呼ばれ、それに対して fork を呼び出した側を親プロセスという。
* Linux の多くのディストリビューションでは、すべてのプロセスは init プロセスの子プロセスである。init プロセスは、カーネルが最初に生成するユーザプロセスである。
* 子プロセスは、ファイル記述子などのプロセスリソースを親プロセスと共有することができる。アドレス空間はコピーが作成され、それぞれ独立して管理される。
* 子プロセスは、exec 系システムコールにより、親プロセスとは別のプログラムに置き換えることができる。
* fork 後は親プロセスと子プロセスの処理は並列して実行されるが、親プロセスは、wait システムコールにより、子プロセスの終了を待つことができる。
* 子プロセスの生成から終了までの一般的な流れ
  - 親プロセス: fork システムコールにより子プロセスを生成
  - 親プロセス: (処理)
  - 親プロセス: wait システムコールにより、子プロセスの終了を待つ
  - 子プロセス: execve により自分自身を他のプログラムに置き換え
  - 子プロセス: (処理)
  - 子プロセス: exit システムコールにより終了
  - 親プロセス: 子プロセスの終了を検知
* 親プロセスと子プロセスは、そのままではアドレス空間は共有しない。これらのプロセスが互いに通信するには、シグナルまたはパイプ、共有メモリあるいはソケットを利用する。

2) プロセスの確認

* 通常/proc にマウントされる proc 疑似ファイルシステムは、現在システム上に存在するプロセスに関する情報を取得するための仕組みを提供する。
* /proc 下にあるプロセス ID ディレクトリ内には、status ファイルなどが存在し、シェル上からプロセス情報を取得するのに便利である。
* /proc 直下には、meminfo や cpuinfo ファイルが存在する。これらはそれぞれ、システムに搭載されている物理メモリ／仮想メモリに関する情報と、CPU に関する情報を含む。
* /proc 下にあるプロセス毎の情報は、top(1)や ps(1)シェルコマンドを使用して一覧表示することができる。

スキル区分	OSS モデルカリキュラムの科目	レベル
システム分野	8　Linux のシステムプログラミングに関する知識 II II	応用
習得ポイント	II-8-5. スレッドプログラミング	
対応するコースウェア	第 10 回 プロセスとスレッド	

II-8-5. スレッドプログラミング

複数プロセスの利用よりも軽量な並列プログラムを実現するスレッドプログラミングについて解説する。プロセスとスレッドの違いについて述べ、pthread ライブラリを利用してスレッドプログラミングを実現する方法を説明する。

【学習の要点】

* アドレス空間はプロセス毎に割り当てられるため、通常複数のプロセス間でアドレス空間は共有されない。
* スレッドは、プロセス内における実行の単位であり、スレッド同士はアドレス空間を共有することができる。スレッドはしばしば軽量プロセスと呼ばれる。
* スレッドはアドレス空間を共有しながら独立にスケジュール可能なため、並列プログラムを実現する際に用いられる。
* pthread は、POSIX によって定義された、Linux 標準のスレッドライブラリである。

	プロセス	POSIX スレッド
生成方法	fork	pthread_create
共有されるリソース	記述子テーブル、メモリマッピングなど	プロセスアドレス空間全体
相互通信	パイプ、ソケット、共有メモリなど各種 IPC	プロセスアドレス空間の参照
主要な同期方法	セマフォ、共有メモリ上のミューテックス	ミューテックス、条件変数

図 II-8-5.　プロセスとスレッド

【解説】

1) プロセスとスレッド

* プロセスとスレッドのどちらも、それを生成した親とは独立してスケジュールが可能である。つまり、親と並列して処理を行なうことができる。
* 子プロセスは、親プロセスとの間でプロセスリソースを共有するが、通常アドレス空間を共有しない（コピーが作成され、それぞれ独立に管理される）。
* あるプロセスにおいてスレッドが生成されると、そのスレッドは親プロセスとの間でアドレス空間を共有することができる。
* スレッドは、プロセス内での実行単位である。このことから軽量プロセス(Light Weight Process [LWS])と呼ばれることがある。

2) スレッドプログラミング

* 複数のプロセスを扱うプログラムと複数のスレッドを扱うプログラムの異なる点は、スレッドの場合はスレッド間でアドレス空間が共有されることである。
* 複数のスレッドが同じアドレス空間にアクセスできるということは、スレッド間で共有リソースに対して適切な同期が必要であることを示唆する。
* スレッド間で共有リソースに対して同期を行なうにはいくつかの方法がある。ミューテックスは最も基本的でよく使用される同期機構である。ミューテックスは、あるアドレス空間にある共有リソースに対して複数のスレッドが同時にアクセスすることを禁止する。

3) POSIX スレッド

* POSIX スレッドは、POSIX の定めるスレッド API である。
* POSIX スレッドは、スレッド使用時に必要とされる、ミューテックスやロックの操作に関するインタフェースも定義する。
* Linux カーネル 2.6 は、POSIX スレッドを使ってスレッド機構を提供する。これは、NPTL (Native POSIX Thread Library) と呼ばれ、GNU C ライブラリに統合されている。
* POSIX スレッドの定める主な関数
  - pthread_create
    スレッドを新しく生成する
  - pthread_exit
    スレッドを終了する
  - pthread_join
    スレッドを親スレッドに合流させる
  - pthread_detach
    スレッドを親スレッドからデタッチする。デタッチされたスレッドは、その終了と同時にメモリリソースが解放される。
  - pthread_self
    現在実行中のスレッド ID を取得する。

スキル区分	OSS モデルカリキュラムの科目	レベル
システム分野	8　Linux のシステムプログラミングに関する知識 II II	応用
習得ポイント	II-8-6. シグナルの利用	
対応する コースウェア	第 11 回 シグナル	

II-8-6. シグナルの利用

シグナルを利用して非同期プログラミングを実現する方法を説明する。シグナルの種類やシグナルの取り扱い方法、タイマー割り込みプログラムの作成方法、プログラムの強制停止とシグナルハンドラ等の概念について述べる。

【学習の要点】

* シグナルとは、プロセス間（またはカーネルからプロセス）で、予め定義されたイベントを送受信する仕組みである。
* プロセスは各シグナルを受信したときのデフォルトの動作を持っている。これを上書きするには、signal システムコールを用いてハンドラ関数を登録する。ただし、プロセスを強制終了させるなどの一部のシグナルについてはこれを上書きすることはできない。
* singal をプロセス間で用いると、互いにプロセス間通信を行うことができる。
* alerm システムコールと pause システムコール、SIGALRM シグナルハンドラを組み合わせると、タイマー割り込みプログラムを作成することができる。

SIGHUP	Hangup	SIGSTKFLT	Stack fault
SIGINT	Interrupt	SIGCLD (SIGCHLD)	Child status has changed
SIGQUIT	Quit	SIGCONT	Continue
SIGILL	Illegal instruction	SIGSTOP	Stop, unblockable
SIGTRAP	Trace trap	SIGTSTP	Keyboard stop
SIGABRT	Abort	SIGTTIN	Background read from tty
SIGIOT	IOT trap	SIGTTOU	Background write to tty
SIGBUS	BUS error	SIGURG	Urgent condition on socket
SIGFPE	Floating-point execution	SIGXCPU	CPU limit exceeded
SIGKILL	Kill, unblockable	SIGXFSZ	File size limit exceeded
SIGUSR1	User-defined signal 1	SIGVTALRM	Virtual alarm clock
SIGSEGV	Segmentation violation	SIGPROF	Profiling alarm clock
SIGUSR2	User-defined signal 2	SIGWINCH	Window size change
SIGPIPE	Broken pipe	SIGPOLL (SIGIO)	Pollable event occurred
SIGALRM	Alarm clock	SIGPWR	Power failure restart
SIGTERM	Termination	SIGSYS	Bad system call

図 II-8-6.　signal.h に記載のあるシグナル

【解説】

1) シグナルの送受信

* シグナルは、あるプロセスに対して、カーネル、他のプロセス、自プロセス、あるいはユーザが特定のイベントを送信するソフトウェア割り込みである。
* プロセスに対して送信されたシグナルは、そのプロセスの用意するシグナルハンドラによって捕捉される。プロセスが明示的にシグナルハンドラを用意しない場合は、デフォルトのアクションが実行される。
* シグナルハンドラを用意することで、プロセスがシグナルに対するデフォルトのアクションを上書き、拒否、または無視することを可能にする。
* カーネルの送信する一部のシグナルは、シグナルハンドラによって捕捉できない。これらは、SIGSTOP と SIGKILL の 2 つのシグナルであり、プロセスの暴走時に強制的に終了させる手段を残すためである。

2) シグナルを使ったプログラミング

* プロセスがシグナルを捕捉するようにハンドラを用意するには、signal システムコールを用いる。signal は、受け取るべきシグナルの種類を示す整数値と、ハンドラ関数へのポインタを取り、以前のハンドラ関数を返す。呼び出しに失敗すると SIG_ERR を返す。
* Linux では、ハンドラ関数ポインタと signal システムコールは、以下のように定義されている。

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

* このとき、ハンドラ関数 handle は以下のようにして signal システムコールに渡すことができる。

```
sighandler_t h = &handle;
signal(SIGBUS, h);
```

* POSIX.1 (IEEE Std 1003.1) 標準のシステムでは、signal よりも一般的な sigaction システムコールを使うことができる。これらのシステムでは signal は sigaction を用いて実装されている。sigaction は、シグナルの生成とハンドラの呼び出しに関してより細かい制御が可能である。
* kill, killpg は、それぞれ他のプロセス、プロセスグループに対してシグナルを送信するのに使うシステムコールである。

3) シグナルを使ったプログラミングの用途

* シグナルはもともと、プロセス間通信を行なうために設計されたものではないが、固定的なイベントを送受信するだけである場合、その目的に使うことができる。
* 実装例として、シグナル SIGALRM の送信と、pause システムコール (POSIX.1 システムでは sigsuspend を使用可能) を組み合わせたタイマーアプリケーションが考えられる。

スキル区分	OSS モデルカリキュラムの科目	レベル
システム分野	8　Linux のシステムプログラミングに関する知識 II	応用
習得ポイント	II-8-7. パイプによるプロセス間通信	
対応する コースウェア	第 12 回 プロセス間通信とパイプ	

II-8-7. パイプによるプロセス間通信

プロセス間通信を実現する手法のひとつとして、パイプを利用した通信手順を紹介する。パイプの作成と入出力制御、名前付きパイプの利用方法などについて、関連する関数を利用してプロセス間通信を実現するための具体的な手順を説明する。

【学習の要点】

* パイプは、一次元バイト列であり、プロセス間でプロセス間通信を行うひとつの方法である。
* パイプは pipe システムコールを用いて生成する。
* パイプで通信できるプロセス同士は、互いに記述子を共有できる間柄（多くの場合、親子関係）になければならない。
* 親子関係にないプロセス同士がパイプを用いて通信を行うには、FIFO（名前付きパイプとも呼ばれる）を使用できる。この場合は、ファイルシステム上のスペシャルファイルを共有することによって同期が取られる。

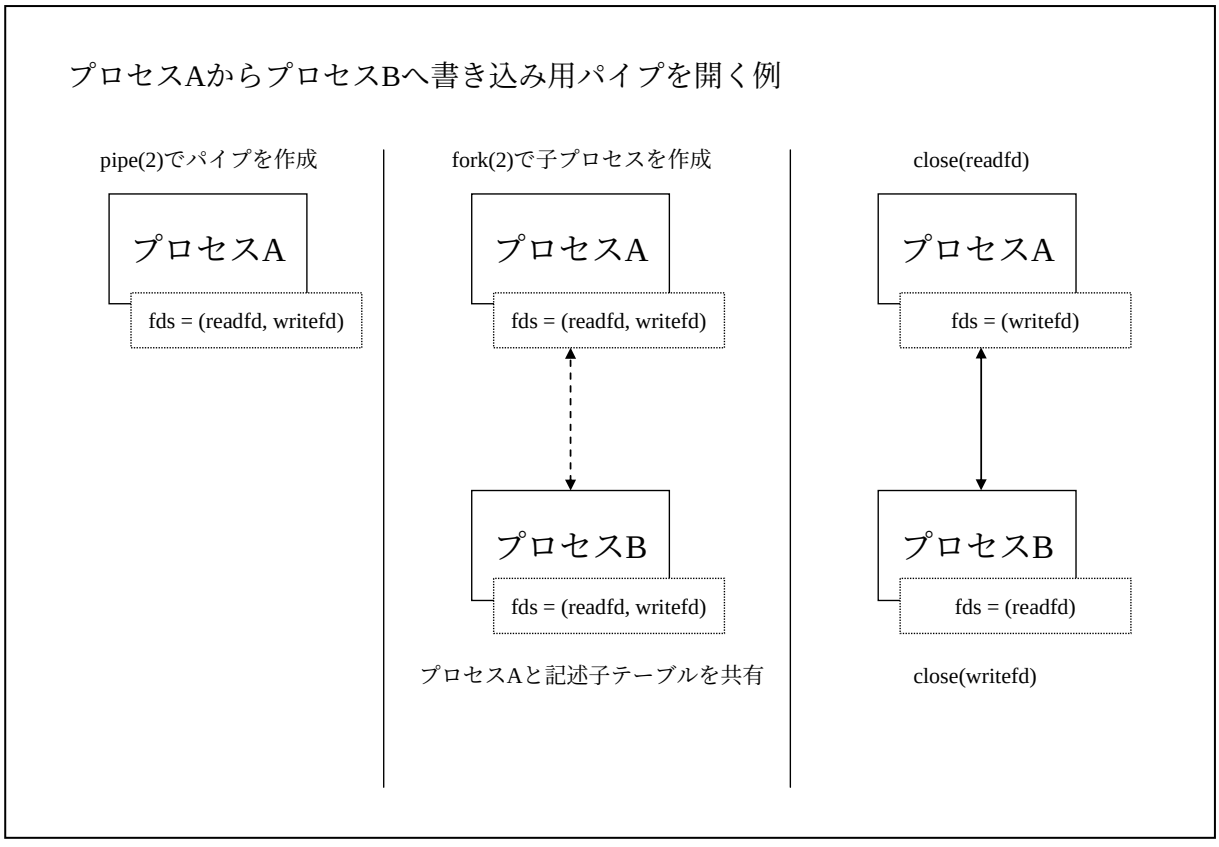

図 II-8-7.　親子プロセス間でパイプを作成

【解説】

1) パイプ

* パイプは、名前を持たないカーネル内のバッファであり、pipe システムコールを用いて作成する。pipe は、2 つのファイル記述子を作成する。それぞれ、読み込み用と書き出し用で、プロセスはこれらの記述子に対する read や write システムコールを通してパイプにアクセスする。作成例を以下に示す。

```
int p[2];
pipe(p);
```

* pipe によって作成されたファイル記述子を、プロセス間で共有することで、2 つのプロセスはカーネル内の同じバッファ領域にアクセスすることが可能である。これは、これら 2 つのプロセスが互いに任意のデータをやり取りできることを意味する。
* fork によって親子関係を持つプロセス同士では記述子の共有が容易であるため、ほとんどの場合パイプはこの状況下で使用される。
* ひとつのパイプは単方向のデータの流れを提供する。これに従って、fork によって記述子を共有するプロセスは、それぞれ一方は呼び込み用記述子をクローズ、他方は書き出し用記述子をクローズする。
* 双方向のデータの流れが必要な場合は、パイプを二つ用意する。
* パイプはシェルで最もよく利用される。これは、読み込み用の記述子を標準入力、書き込み用の記述子を標準出力にそれぞれマップすることで行なう。

2) FIFO

* FIFO (First In, First Out) は、パイプに似た機能を提供する特殊ファイルである。スペシャルファイルであることから、これはファイルシステムの名前空間上にマップされる。このため、これを名前付きパイプと呼ぶことがある。
* FIFO は、mkfifo システムコールを用いて作成する。

```
mkfifo(“/tmp/myfifo”, mode);
```

* パイプと違い、ファイルシステムのパスと関連付けられる（名前を持つ）ため、関係のないプロセス同士が名前で同一 FIFO にアクセスすることができる。

3) パイプ、FIFO に共通の性質

* パイプも FIFO もカーネル内に置かれるバッファである。バッファの長さはシステムに依存し、POSIX.1 では最低 512 バイトを要求する。
* パイプや FIFO のサイズに収まるデータを write によって書き込んだ場合、write は不可分性を保証する。もし、パイプや FIFO のサイズ以上のデータを書き込んだ場合にはこれは保証されない。
* パイプや FIFO は、何もしなければブロッキングモードのファイル記述子を使用することになる。fcntl を使用して記述子を非ブロッキングモードに設定することで、パイプまたは FIFO の処理を非ブロッキングモードで実行できる。

スキル区分	OSS モデルカリキュラムの科目	レベル
システム分野	8　Linux のシステムプログラミングに関する知識 II II	応用
習得ポイント	II-8-8. デバイスファイルの扱い方	
対応するコースウェア	第 13 回 端末機器の入出力	

II-8-8. デバイスファイルの扱い方

キーボード、ディスプレイ、プリンタ、ディスクといった全てのデバイスをファイルとして一元的に取り扱う考え方を説明する。また、入出力先を変更するリダイレクトについて述べ、デバイスファイルを簡単に利用する方法を紹介する。

【学習の要点】

* 各種ハードウェアデバイスは、カーネル内のデバイスドライバによって操作される。ユーザは、スペシャルファイル（デバイスファイル）を通してカーネル内のデバイスドライバの機能にアクセスできる。
* スペシャルファイルは、カーネルによって特別に扱われる一方、ユーザからは通常のファイルのように見える。このためプログラムからは、open, read, write などのシステムコールを用いて記述子を取得し、アクセスすることができる。
* ある記述子に関連づけられたストリームを、別の記述子に関連づけることを入出力リダイレクションと呼ぶ。通常標準出力に出力されるストリームをファイルに出力するには、端末デバイスの記述子 1 を閉じ、続いて目的のファイルを開いて新たに記述子 1 を獲得することで行われる。

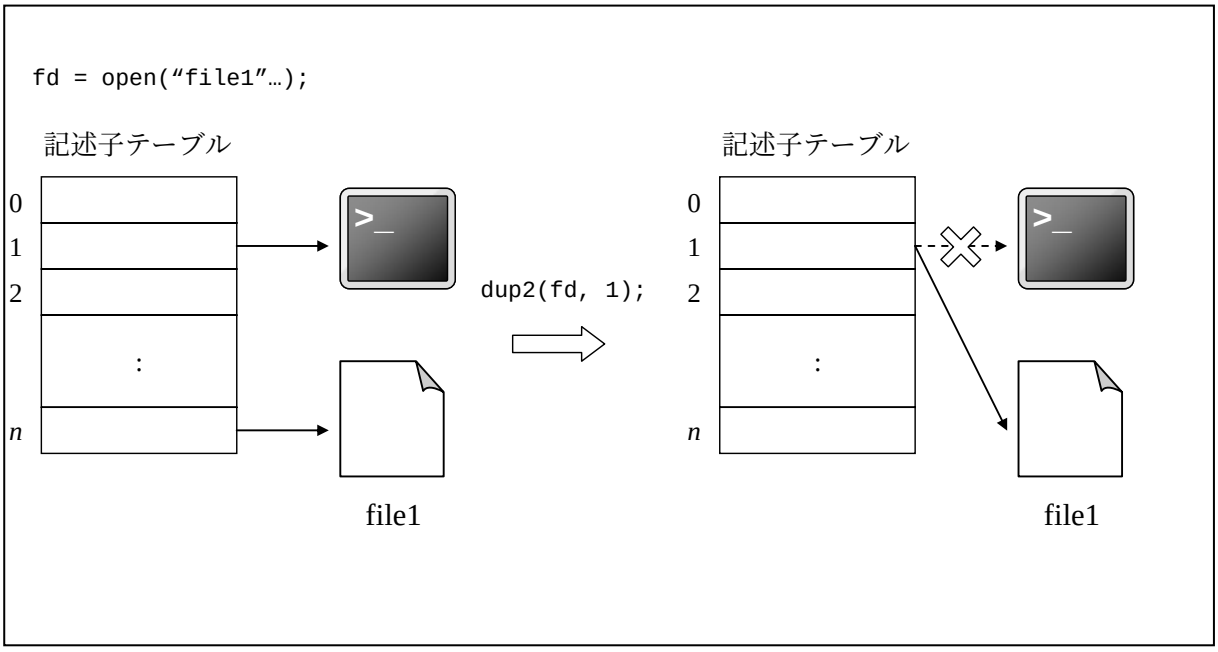

図 II-8-8.　dup2 によるリダイレクト

【解説】

1) デバイスファイルの扱い方

* Linux カーネルは、ハードウェアデバイスに対するデバイスドライバを実装する。
* デバイスファイルは、ユーザプロセスが、カーネル内のデバイスドライバにアクセスすることができるようにカーネルが提供するインタフェースである。これはスペシャルファイルとも呼ばれる。ファイルシステム上に名前を持ち、ユーザプロセスは通常のファイルと同じ read, write などのシステムコールを用いてアクセスできる。
* デバイスドライバ側は、ユーザプロセスからのシステムコールを受けて、それらに対応するオペレーション関数を実装している。この中には、通常のファイルに対しては使用しないような、ioctl, poll システムコールに対応するオペレーション関数も含まれる。

2) デバイスファイルを使ったプログラミング例

* Linux には、/dev/rtc というハードウェアクロックを制御するためのキャラクタデバイスファイルが存在する。このデバイスファイルを使ったプログラミングの例を示す（エラー処理は省略してある）。
* /dev/rtc は、デバイスファイルなので open システムコールでオープンすることができる。ファイル記述子を得る。

```
int rtcfd;
rtcfd = open("/dev/rtc", O_RDONLY);
```

* ioctl システムコールは、主にデバイスを設定するのに使用される。この例では、ioctl で割り込み周期（tick）を 8192 に設定する。

```
ioctl(rtcfd, RTC_IRQP_SET, 8192);
```

* ioctl で制御できる項目と引数の型は、デバイスドライバ毎に異なる。通常デバイスドライバのヘッダファイル（この例では、/usr/include/linux/rtc.h）で知ることができる。
* read システムコールで、デバイスファイルの内容を読む。この例では割り込みを待つ（読み込み可能になるまでブロックされる）。値を読み込むことができると read が返る。

```
unsigned int data;
read(rtcfd, &data, sizeof(data));
```

* デバイスファイルの使用を終えたら、close システムコールでクローズする。

```
close(rtcfd);
```

3) デバイスファイルの入出力リダイレクト

* シェルのリダイレクトを利用して、デバイスファイルに対して通常のファイルのように、入出力先を変更することができる。例えば以下の例は、シリアル端末（/dev/ttyS0）からの情報をターミナルエミューレータの画面で表示する入力リダイレクトの例である。

```
cat < /dev/ttyS0
```

スキル区分	OSS モデルカリキュラムの科目	レベル
システム分野	8　Linux のシステムプログラミングに関する知識 II II	応用
習得ポイント	II-8-9. セマフォ、共有メモリ、メッセージキューの利用	
対応するコースウェア	第 14 回 セマフォ、共有メモリ、メッセージキュー	

II-8-9. セマフォ、共有メモリ、メッセージキューの利用

プロセス間通信で必要な手段であるセマフォ、共有メモリ、メッセージキューについてそれぞれの概念を理解させる。セマフォを利用した排他制御、共有メモリを利用したプロセス間のデータ転送、メッセージキューを利用したプロセス間通信の具体的な方法を解説する。

【学習の要点】

* プロセス同士が通信する際には、一方のプロセスが、他方のプロセスの処理の完了を待つ必要がある場合がある。これを、互いのプロセスの同期を取るという。
* プロセスを同期する代表的な手法に、セマフォを用いる方法がある。よく使われるバイナリセマフォは、0 と 1 の状態を持ち、プロセスが処理を続行可能であるか否かの判断を行うために用いられる。
* プロセスはそれぞれが独自のアドレス空間を持ち、原則として互いのメモリ領域に踏み入ることはできない。例外として、共有メモリは二つ以上のプロセスがアドレス空間を共有することを可能にする。
* メッセージキューはカーネルによって保持されるキューで、プロセスやスレッドはメッセージを置いたり取り出したりといった操作を非同期で行うことができる。

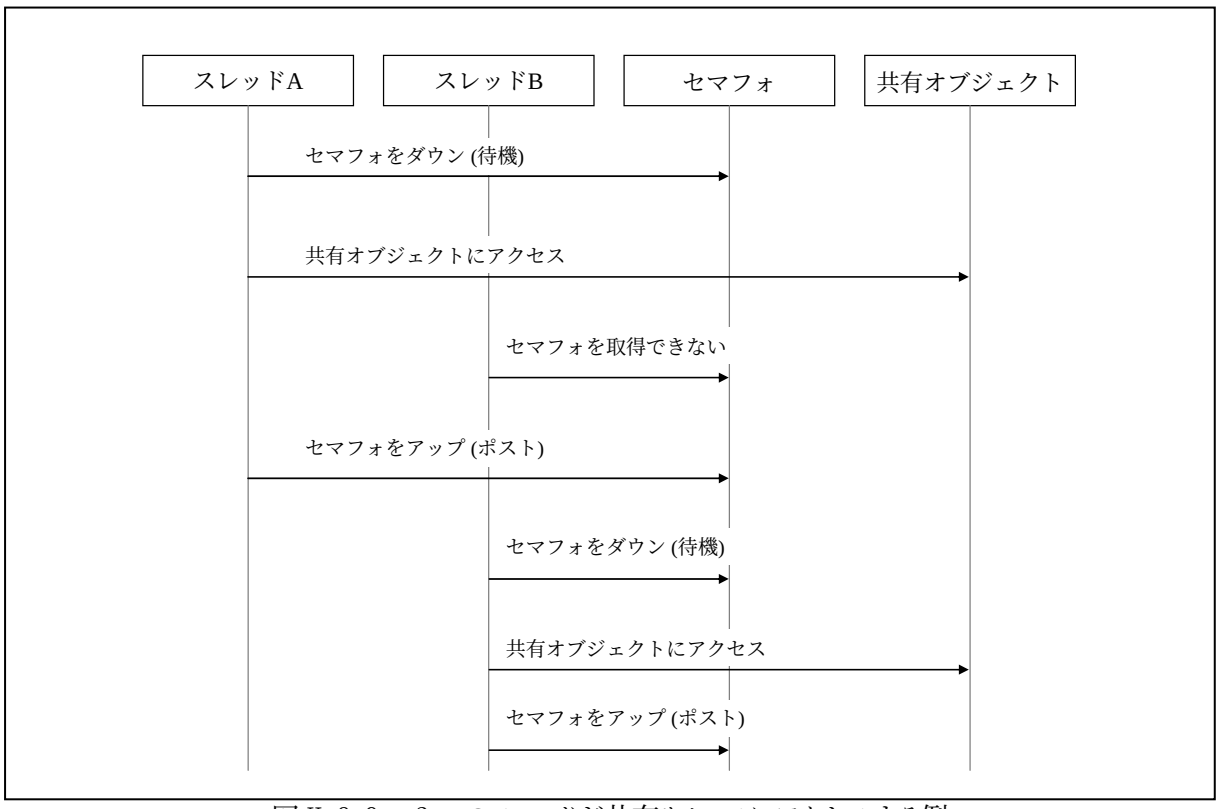

図 II-8-9.　2 つのスレッドが共有リソースにアクセスする例

【解説】

1) 同期とセマフォ

* 複数のプロセス間、またはスレッド間でデータを共有する場合、一方のプロセスまたはスレッドがデータの内容を変更している間、他方のプロセスまたはスレッドはその変更の完了を待つことが必要である。このことを同期をとるという。
* Linux は複数の同期手段を提供する。ミューテックスと条件変数を使った同期、セマフォを使った同期が比較的よく利用される。
* セマフォを使った同期の基本的な概念は、カーネル内またはファイルシステム名前空間上に存在する単一のセマフォを、複数のプロセスが監視することである。この流れは概ね以下の通りである。
  - プロセス 1 と 2 は、セマフォへの参照を作成する。
  - プロセス 1 が、自分の処理が続行可能かどうかをセマフォに問い合わせる。
  - セマフォはプロセス 1 の処理の続行を許し、それを記録する。
  - プロセス 2 が自分の処理が続行可能かどうかをセマフォに問い合わせる。
  - セマフォはプロセス 1 が処理中にプロセス 2 の処理の実行を許すかどうか決定する（このポリシーはセマフォの種類により異なる）。
  - セマフォが処理の続行を許せばプロセス 2 は処理を続行する。そうでなければプロセス 1 の処理の完了を待つ。

2) 共有メモリ

* ひとつのプロセス内の複数のスレッドは、それぞれがプロセスのアドレス空間を共有することができるが、異なるプロセス同士は通常これができない。
* プロセスのアドレス空間は、通常自身のプロセスからのみ参照を許すが、共有メモリの仕組みを利用すると、プロセスは自分のプロセス空間のある範囲を他のプロセスに公開することができる。
* 共有メモリ領域は、参照を許されたそれぞれのプロセスから直接に（カーネル内バッファを経由せずに）読み書きできるため、最も高速なプロセス間通信といえる。
* それぞれのプロセスは並列に動作するにも関わらず共有メモリ領域を参照できるので、この領域を不正に変更されることのないよう適切な同期をとる必要がある。

3) メッセージキュー

* メッセージキューは、リンク構造を持ったカーネル内の構造体であり、非同期のプロセス間通信を実現する方法である。
* あるプロセスがメッセージキューにメッセージを置くと、構造化されたデータ（レコード）としてカーネル内に保持される。次に別のプロセスは、このレコードを読むことができるようになる。
* それぞれのメッセージには優先度を設定することができる。
* メッセージを読む側のプロセスが、キュー内のメッセージの有無を知る方法は、ブロックして待つ（同期をとる）か、ブロックせずにポーリングするか、シグナルによって通知を受けるかのいずれかである。

スキル区分	OSS モデルカリキュラムの科目	レベル
システム分野	8　Linux のシステムプログラミングに関する知識 II II	応用
習得ポイント	II-8-10. ソケットによるネットワーク通信	
対応する コースウェア	第 15 回 ネットワークプログラミング	

II-8-10. ソケットによるネットワーク通信

ネットワークを超えた通信を可能とするソケットの概念を解説する。さらにソケットを用いたネットワーク通信の具体的な手順を示し、サーバプログラムの簡単な動作やホストバイトオーダとネットワークバイトオーダの違いといった関連する話題についても触れる。

【学習の要点】

* ソケットは、ネットワークを超えてプロセス同士が互いに通信を行うことを可能にする。
* ソケット API は扱うネットワークに依存しない。カーネル内で定義されているドメインを指定することで扱うネットワークを指定できる。
* ソケットは、socket システムコールにより生成され、bind システムコールによりドメイン内で有効な名前を付与する。
* バイトオーダとは、複数バイトを保存する際に、1 バイトずつをどういう順序で保存するかを決定するものである。ネットワークを介して通信する際にはネットワークバイトオーダを使用する。

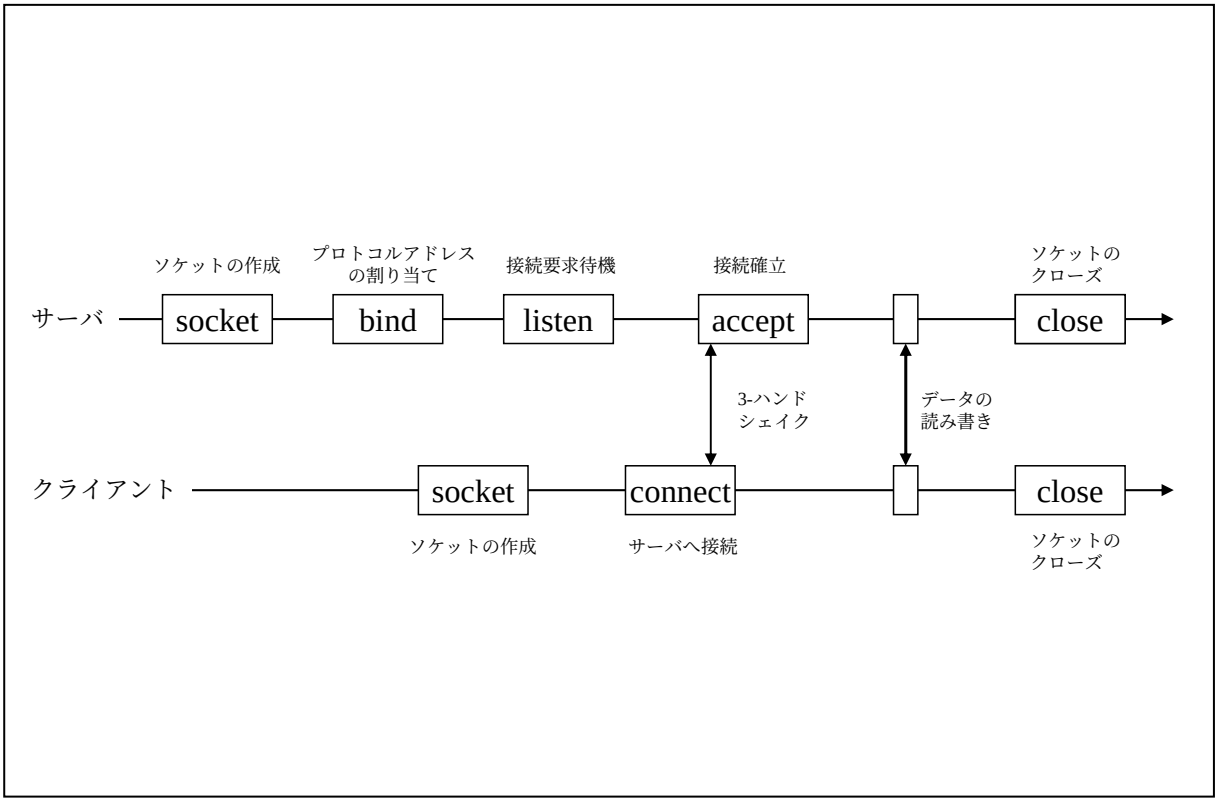

図 II-8-10.　TCP サーバとクライアントの接続

【解説】

1) ソケット API

* ソケット API は、ホスト内のプロセス間通信だけでなく、ネットワークを越えてホスト間で通信を行なえるように設計されたものである。
* ソケット API はネットワークのプロトコルに依存しないように設計されている。Linux では、UNIX ドメイン、INET ドメイン、BLUETOOTH ドメインなどがソケット API を使用する。
* ソケット API で「ソケット」とは、通信の端点（エンドポイント）を意味する。ソケットは、socket システムコール（厳密には Linux ではシステムコールではない）を用いて作成することができる。
* エンドポイントを識別するために、ソケットにはプロトコルアドレスが割り当てられる。プロトコルアドレスは sockaddr 構造体で総称（抽象化）され、その実装はプロトコルに依存する。
* ユーザプロセスは、ソケット API の提供する一連の関数を用いてカーネルとデータをやりとりする。カーネルは、下位のネットワークサブシステムを呼び出し、実際の通信を行なう。

2) TCP ソケット

TCP 通信を実現することは、おそらく最もよくあるソケット API の利用例である。TCP 通信を行なうための使用する主な関数は以下の通りである。

- socket

  システムコールによりソケットを作成する。このとき TCP プロトコルでの通信を行なうよう、適切な引数を指定する。

  ```
  int sock = socket(AF_INET, SOCK_STREAM, 0);
  ```

- connect

  TCP クライアントがサーバに接続する際に用いる。ソケット、サーバの TCP プロトコルアドレス（IP アドレスとポート）を指定する。

  ```
  connect(sock, (struct sockaddr *) &srv, sizeof(srv));
  ```

- bind

  ソケットにプロトコルアドレスを割り当てる。通常クライアントではこれを省略する。サーバでは、自身のローカルアドレスを割り当てる。

  ```
  bind(sock, (struct sockaddr *) &local, sizeof(local));
  ```

- listen

  TCP サーバがクライアントの connect 要求を待つ。サーバは接続してきたクライアントをキューに入れて保持する。

  ```
  listen(sock, 5);
  ```

- accept

  TCP サーバが接続確率済みキューからクライント接続を取り出す。

  ```
  accept(sock, (struct sockaddr *) &client, sizeof(client));
  ```

3) バイトオーダ

バイトオーダとは、複数バイトを保存する際に、1 バイトずつをどういう順序で保存するかを決定するものである。ホストによってバイトオーダは異なる場合があるため、ネットワークを介して通信する際には、一貫してネットワークバイトオーダを使用する。