ゼミ: Software Design Methods for Concurrent and Real-Time Systems

久野　靖*

1999.9.24〜

□ 本読みゼミ…目的は?

- 英語を読む訓練 (だから自分のパート以外もちゃんと読んでくること!)
- まとまった内容を体系的に学ぶ (cf.　新しい内容→論文を読む)
- 内容をネタに議論?雑談?　する
- 旧交を暖める?　新しい知合いを作る?

1　本について

□ Hassan Gomaa, Software Design Methods for Concurrent and Real-Time Systems, Addison-Wesley, 1997.

- 中谷さんのお勧め
- 設計技法という考え方を知らない人には知って欲しいので
- リアルタイムシステムについての勉強になる
- ソフトウェア工学の概観を得るのにもそれなりに良い?

2　各自の Duty について

□ 担当部分を紹介する

□ 資料は必ず用意する

- 全翻訳しようとする人がいるがそれはダメ

□ OHP はできれば用意する

- まあ皆様忙しいので…

□ 重要!　分からないことは分からせてから来る

- 質問はメールでいつでもください

3　はじめに

□ マイクロプロセッサ (μ P) が廉価となり普及

- → (分散) 実時間システムが多くの問題に対する有効な解答に
- →軍用/医療/消費者むけシステムの多くがμ P ベースに

□ 本書は並行システム (concurrent system) の設計をテーマとする

- 並行システムのうちの重要なカテゴリである (1) 実時間システム、(2) 分散アプリケーションについても特に取り上げる

□ 第 1 章→並行システム、実時間システム、分散アプリケーションが持つ特性について

3.1　並行性

□ 昔:バッチシステム→現在:対話型システム+μ P →並行システム

□ 並行システム→複数の動作が並列に発生する→入力 (イベント) の順序は予測不能かつ同時に起こることもあり得る

3.2　並行タスク

□ タスク→逐次的なプログラムの実行、ないし並行プログラム中の逐次的な成分

- タスクの内部には並行性はない
- 複数のタスクが並列実行→システム全体としての並列性
- 各タスクは通常、非同期的に実行し、ほとんどの時間は互いに独立に実行している (必要に応じて通信/同期)

□ 複数の並行タスクの集まり→さまざまな知見が得られている

*筑波大学大学院経営システム科学専攻

- E. W. Dijkstra →理論的研究
- Per Brinch Hansen →並行タスクに基づく OS ←セマフォ、メッセージ
- C. A. R. Hoare →モニタ←同期に対する情報隠蔽
- 多くのアルゴリズム→例：読み手と書き手、眠れる床屋、哲学者の食事、銀行家

□ 並行タスクの考え→ OS、DBMS、実時間システム、対話型システム、分散システム、シミュレーションなどに応用

- 並行システム設計の鍵→ (1) システムを並行タスクの集合で表し、(2) それらを相互通信させ、(3) 同期を取らせ、(4) 共有データを安全にアクセスさせる
- さらに、プログラミング言語や OS によるサポートも重要

3.3 並行タスクの利点

□ 1．アプリケーションにとって自然なモデル←問題領域に内在する並列性を素直に表現

- このようなアプリケーションでは並列性を前提とした記述が最適→逐次的な仕様より明確で理解しやすい←問題領域を素直に記述

□ 2．並行システムをタスクに分割→各タスクが「何をやるか」「いつやるか」を分離できる→理解しやすく管理しやすい構造に

□ 3．システムを並行タスクの集まりで構成→実行時間の短縮← I/O と計算の並行動作、マルチ CPU の活用

□ 4．並行タスクの集まり→スケジューリングの自由度が大→実時間制約の強いタスクを優先できる

□ 5．設計の早い段階で並行タスクへの分解→早い段階で性能分析が行える←多くのツールや分析手法

□ ただし：あまりに多くのタスク→複雑さ、通信/同期のオーバヘッド増大

3.4 実時間システム

□ 実時間システム→入力イベントに対して応答 (出力イベント) →多くのシステム (産業用、消費者むけ、軍用) で使われる

- 難しさ→イベントの順序やタイミングは予測不能→それらに対し仕様通りの応答を返す必要、さらに負荷が大幅に変動することも

□ 多くの実時間システムは並行システム←さまざまな外部事象が並行して発生し、それに応答する

- 実時間システムとは→「時間制約を持つ並行システム」
- ハードリアルタイム→厳密な時間制約 (デッドライン) を守る必要
- ソフトリアルタイム→ときどきデッドラインを守り損なっても許容できる

□ 実時間システムが持つ、他のシステムにない特徴：

□ 1．組み込みシステム：他のハードウェア/ソフトウェアシステムに組み込まれたシステム

- 例：ロボットコントローラ→サーボ機構、センサ、アクチュエータを制御
- 例：クルーズコントロール→自動車の速度を制御

□ 2．外部イベント (人間以外の入力) に応答：センサの入力などを持つ

- 例：機械加工制御、化学反応制御など

□ 3．時間制約：対話型システム (人間が入力) →応答が遅れてもいらいらする程度。実時間システム→応答が遅れると重大

- 例：航空管制システム→応答が遅れたら空中衝突に…
- 時間制約のオーダー→ミリ秒～秒～分単位のことも。

□ 4．実時間制御：人間の判断の介在なしに制御の判断を行う。

- 例：クルーズコントロール→人間が判断しなくても速度を制御
- 実時間システムの中に実時間で動作しなくてもよい要素があってもよい。例：実時間でデータ計測→データの保存はゆっくりでよい

□ 5．応答型 (reactive) システム：イベントドリブン型で外部からの刺激に反応を返す。応答はシステムの状態に応じて変化し得る。

3.5 分散アプリケーション

□ 分散アプリケーションとは→並行アプリケーションで、なおかつ複数の場所にある複数のノードを含む環境で実行されるもの。

- 各ノードは別個の計算機システム→相互にネットワークで結合
- システムソフトウェアに多い→分散 OS、分散 DB、分散ファイルシステムなどが代表。本書では一般のアプリケーションを対象。

□ 分散処理の利点:

□ 1．可用性 (availability) の増大: 一部のノードがダウンしても実行を継続できる

□ 2．柔軟な構成: アプリケーションをさまざまな配置で実行できる

□ 3．局所的な管理/制御: 各ノードが自律的に動作するように設計できる

□ 4．段階的増強: 負荷が大きくなったらノードを増やして対応

□ 5．低コスト: μ P のコストダウンのおかげで、集中型システムより廉価になることが多い

□ 6．負荷分散: アプリケーションによっては負荷をノード間で分担できる

□ 7．応答時間の改善: ローカルに処理できる応答は即時的にできる

4 並行/実時間システムとソフトウェアライフサイクル

□ あらゆるソフトウェアシステム同様、並行/実時間システムもライフサイクル (=フェーズに基づくやり方) に従って開発されるべき

- 代表的なライフサイクルモデル: 滝モデル
- 滝モデルの弱点に対処→使い捨てプロトタイプ、漸進開発 (発展型プロトタイプ)、スパイラル
- 検証と証明も重要

4.1 滝モデル

□ 過去 20 年間→ソフト開発コストは上昇、ハードのコストは下降→典型的なプロジェクトの全コストの 80%がソフトのコストに

- 1960's →ソフト開発の何が問題かはよく知られていなかったが「ソフトウェア危機」であることは認識されていた→「ソフトウェア工学」のはじまり
- ソフトウェア工学→大きなソフトウェアシステムを効果的に開発するのに必要な管理手法、技術的手法、手順、ツールを研究する分野
- →多くのソフトウェアがライフサイクルに基づいて開発された
- →このライフサイクル=滝モデル。以下の段階に基づく

4.2 要求分析/仕様

□ ユーザの要求を認識し分析→ソフトウェア要求仕様 (SRS) を生成

- →システムが「何をするか」を定める→「どうやってやるか」は定めてはならない

□ 実時間ソフトは大きなシステムの一部であることが多い→システム要求仕様が既に存在してそこから SRS を起こすことに

4.3 概要設計

□ システムをその構成要素に分解→並行/実時間システムの場合は特に、並行タスクへの分解が重要

- 設計手法/設計者の判断により、タスクへの分解/モジュールへの分解のいずれが重視されることもある
- システムのふるまいの側面を設計することも重要

4.4 詳細設計

□ 各要素の動作手順をプログラム設計言語 (PDL)---疑似コードのこと--- で記述

- 並行/実時間システムでは資源共有、デッドロック回避、I/O へのインタフェースの設計も重要

4.5 コーディング

□ プログラム言語による記述を行う

- 並行システムの場合、並行機能を持つ言語（Modula-2、Ada）を使うことも直列言語+OS等の機能呼び出しを用いることもある

4.6 ソフトウェアテスト

□ 並行/実時間システムだからといって他のシステムのテストと変わらない部分も多い→違うとすれば並行タスク、I/O関係の部分

□ 主な問題はシステムの動作が非決定的であること→これを決定的な形でテストする方法も提案されている

□ 組み込みシステムの場合はさらに面倒→テストのためにシミュレータが必要な場合も→さらに性能のテストも必要

□ 通常いくつかの段階に分けてテスト←誤りの検出やその場所の決定が難しいことから来ている

- 単体テスト、統合テスト→ホワイトボックスアプローチ
- システムテスト→ブラックボックスアプローチ

4.7 単体テスト

□ 各部品を単体でテストする→最小限、文カバレジ、分岐カバレジ

4.8 統合テスト

□ 部品群を順次組み合わせて行き、最後はシステム全体として動作できるように

□ 並行システムの場合、並行タスク間のインタフェースをテストすることが特に重要→第18章で扱う

4.9 システムテスト

□ システムをすべて組み立てて、要求仕様に照らしてテスト→独立したテストチームが行うことが望ましい

- 統計的利用テスト→予想利用形態に基づきテストシナリオを生成してテストを行う

□ 並行/実時間システムのテストの内容:

□ 1．機能テスト：システムが要求仕様に規定されたように動作することを確認

□ 2．負荷テスト：実運用で発生すると思われる負荷や負荷変動に耐えることを確認

□ 3．性能テスト：システムが時間制約を満たすことを確認

□ 実時間システムのテスト→環境シミュレータによって外部環境を何回でも再現可能にできるとよい

4.10 受領テスト

□ ユーザないしその代表によって実施→最終的にシステムを受理するかどうかを決定→システムテストに類似

4.11 滝モデルの限界

□ 滝モデルはそれ以前の無秩序に比べて大きな進歩→しかし実際には「滝」といいつつフェーズ間のオーバラップや反復は必要→しかしそれでもまだ大きな弱点:

□ 1．要求仕様は最後にシステムが動作してユーザに見せるまで適切なテストがなされない→「要求仕様の誤りはすべての誤りのうちで一番最後にみつかり、修正に要するコストも最大である」

□ 2．動作するシステムはライフサイクルの最後になってやっと現れる→その段階まで設計上/性能上の問題が看過されがちで、その段階ではもう手直しできない

□ これらによるリスクが大きい場合（例：要求が明確でない）→滝モデルの弱点を改良する方法→使い捨てプロトタイプ（上記1を改良）、漸進的プロトタイプ（上記2を改良）

4.12 使い捨てプロトタイプ

□ 使い捨てプロトタイプ→短期間/低コストで動作するシステムを開発し、それをユーザに見せることで要求仕様を明確にする

- →初期の要求仕様が出た段階で作成できる
- →ユーザに動くものを見せることで多くのフィードバックが得られる
- →フィードバックに基づき要求仕様を改良→あとは普通に開発

□ そのほか、実験目的 (アルゴリズムの確認、性能の確認) のために使い捨てプロトタイプを作ることもある

□ 使い捨てプロトタイプは対話型システムの要求仕様を固めるのに特に効果的←ユーザと開発者のコミュニケーションバリアを克服

4.13 発展型プロトタイプ/漸進的開発

□ プロトタイプを順次改良していって最終製品に至る→システムの重要部分が時間制約を満たすことを確認しつつ開発→リスクの低減

- 各段階で追加する部分の選別→イベントシーケンス図などを使用

□ 最終製品の一部分が初期の段階からプロトタイプとして動くところに特徴がある

□ 初期の段階から動く部分がある→モラルの向上、制約をクリアすることの確認、システム統合を時間的に分散

4.14 使い捨てプロトタイプと漸進的開発の組み合わせ

□ 発展型プロトタイプは最終製品の一部となるのでそれなりの品質を持って開発する必要がある

□ まず使い捨てプロトタイプで要求仕様を確認してから漸進的開発を使うという選択肢もある

4.15 スパイラルモデル

□ らせん状に反復するプロセス→各反復はリスクアセスメントを行う

- 各反復の中では仕様検討→プロトタイプ作成→評価
- 反復の結果十分リスクが低減されたら本番実装に入る

4.16 設計の証明と検証

□ 検証 (validation) →システムがユーザの要求に合致することを確認

□ 証明 (verification) →実装が仕様に合致することを確認

□ ソフトウェア品質保証 (SQA) →ソフトウェア製品の品質を保証する活動全般を指す。例: レビュー

□ 検証の手法→使い捨てプロトタイプ、システムテスト

□ 証明の手法→レビュー、プロトタイプ、シミュレーション

5 ソフトウェア設計の諸概念

□ 本章では並行/実時間システムにおける重要な概念を整理

- 並行プロセスの概念 (1 章より詳しく)
- システム環境、OS 環境
- 情報隠蔽
- オブジェクト指向、クラス、継承
- 有限状態マシン (FSM)