# 教育用OSのソフトウェア工学的側面 ―udosの開発を通して―

**Software engineering aspects of educational operating systems – through the development of udos –**

**権藤 克彦*　大場 勝†**

**Summary.** Educational operating systems are designed to give undergraduate students "hands-on" experience with the essense of OSes. Interestingly, most educational OSes are just designed and implemented as small as possible, and there are few software engineering discussions on policies and tradeoffs for them. In this paper, we provide some software engineering aspects of educational OSes gained through the development of udos.

## 1　はじめに

OS とはコンピュータに奇跡を起こさせる魔法であり，OS の講義ではその魔法の秘密を学生に説明する必要がある [5]．しかし，学部レベルの OS の講義では，プロセスなどの抽象的な概念や，プロセススケジューリングなどのアルゴリズムに偏りがちなため，OS の魔法の秘密を十分に解き明かせないことが多い．この問題を解決するために教育用 OS が提案されており，MINIX [2]，TUNIX [7]，Nachos [4]，PortOS [3]，Topsy [8]，GeekOS [6] など数多くある．Linux などの実用的な OS は複雑で巨大すぎるので教育には向かない．そこで教育用 OS では複雑な OS の内部実装を，その本質を保存しつつ，理解・修正しやすくすることを目標にしている．

興味深いことに，提案されている教育用 OS のほとんどは，実装規模を小さくすることに主眼を置いているが，その際の設計上の方針やトレードオフに対するソフトウェア工学的な議論はほとんどされていない．特に「なぜ，抽象的な概念やアルゴリズムだけではダメなのか? 教育用 OS は何を補完しているのか?」という疑問に答えていない．本論文では，この疑問に答えて，ソフトウェア工学的に興味深い題材を提供することを試みる．我々の答えの要約は以下である．

- OS の本質は，抽象概念を実現する OS とハードウェアの役割分担にある．
- OS 中の抽象概念ではこの役割分担を説明できない．
- 教育用 OS は役割分担の追跡性 (traceability) に優れている (べきである)．

役割分担の理解が重要なのは，OS とハードウェアが協調して，デバイスドライバ，プロセススイッチ，仮想メモリなどを実現するので，ハードウェアの機能を理解した上で OS との役割分担を理解しないと，OS の本質を理解できないからである．しかし，多くの学生はハードウェアの知識がないため，役割分担を理解できない．

本論文では，まず OS の抽象化を議論する (2 節)．OS の抽象概念は重要ではあるが，OS 中の抽象概念や半同期/半非同期パターン [23] では，この役割分担を説明できないことを述べる．次に 3 節では，udos [25] を例にとり，教育用 OS ではこの役割分担を説明しやすいこと示す．udos は我々が開発した教育用 OS である．

*Katsuhiko Gondow, 東京工業大学 情報理工学研究科 計算工学専攻

†Masaru Ohba, 東京工業大学 情報理工学研究科 計算工学専攻

## 2　既存の抽象化は OS とハードウェアの役割分担を説明しない

一般的に，抽象化とは「性質や機能の一部を意図的に無視して詳細を省き，重要な点を残すこと」を指す．抽象化により実装の細部の隠蔽やインタフェースの共通化ができるため，抽象化のソフトウェア工学的価値は高い．本節では，OS 教育に適した抽象化とユーザや OS/デバイスドライバ開発者に適した抽象化とは異なることを述べる．また OS 教育に適した抽象化やモデルがないことも述べる．

### 2.1　OS 中の抽象化

OS では，大きく次の 2 種類の抽象化が存在する．

- **ユーザ**のための抽象化— 例えばプロセスやファイルがこれに相当する．ユーザは `fork` や `open` などのシステムコールを介して，この種類の抽象物を操作する．これは OS が提供するサービスの抽象化なので，OS の外部仕様とも言える．
- **OS 開発者**や**デバイスドライバ開発者**のための抽象化— UNIX の仮想ファイルシステム (VFS)，STREAMS，デバイススイッチ (devsw) や Linux の LKM(loadable kernel module) などがこれに相当する．これは基本的にはインタフェースの統一化であり，OS の拡張性・移植性・保守性を高くする．

OS 中の抽象化の理解は重要だが，残念ながら，どれも OS とハードウェアの役割分担を理解するための抽象化ではない．外部仕様では内部仕様を理解できないし，インタフェースの統一化は間接層を増やすため，むしろ理解を難しくするからである．

### 2.2　半同期/半非同期パターン

半同期/半非同期パターン[1]は OS の入出力操作で典型的に用いられる手法である．下位レベル (カーネル空間に相当) では非同期に効率よく入出力を行いつつ，上位レベル (ユーザ空間に相当) では同期モデル (`read` などのシステムコールに相当) を提供してプログラミングを容易にする (図 1)．半同期/半非同期パターンはユーザ空間とカーネル空間の役割分担に焦点があてられており，ハードウェアと OS の役割分担を十分には表現していない (単に割り込みを使うとしか述べられていない)．ハードウェア上の制約や OS が割り込みを待つ際にプロセススイッチする必要があることはこのパターンからは読み取れない．

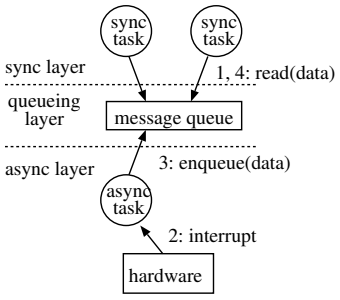


**図 1　Half-sync/half-async pattern**

## 3　教育用 udos における半同期/半非同期パターン

### 3.1　教育用 OS udos

udos [25](図 2) は我々がスクラッチから開発した教育用 OS である．主な特徴は次の通り．

- 学生が試しやすいように udos は PC/AT 互換機上で動作する．また，HDD にはアクセスせず FDD のみで動作するのでハードディスクへのインストールは不要．
- 開発効率向上のため，udos 開発に PC/AT 互換機エミュレータ Bochs [1] を使う．

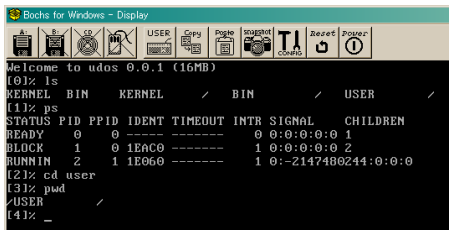


**図 2　udos running on Bochs**

[1] ここでは OS での入出力の抽象化の一例として半同期/半非同期パターンを取り上げた．他にも OS とハードウェアの役割分担が重要な場合がある．例えば，ページフォルトハンドラと MMU など．

- カーネルは非常に小さい．そのために udos は必要最小限のハードウェア[2]だけをサポートしている．udos は約 5,000 行の C と少しのアセンブリコードで書かれており，非常に小さい．ちなみにカーネルはモノリシックである．
- 怪しげなノウハウでなく，CPU やハードウェアの仕様書に基づいて開発した．例えば，書籍 [16] [17] だけでなく， [18] [19] などの仕様書も参照した．関連する仕様書は膨大になるが，我々が開発したドキュメンテーションツール ADIOS [24] での解決を現在試みている．
- その他：IA-32 の保護モード・仮想メモリを使用．preemptive マルチタスク (ただしカーネルは nonpreemptive)．FAT12 ファイルシステム．UNIX ライクな API．Cygwin 上でカーネルとユーザプログラムをクロスコンパイル可能．

### 3.2 udos の半非同期部分の実装

udos のフロッピー制御コード (fd.c) はバッファキャッシュを含めて約 500 行であり，半同期/半非同期パターンになっている．図 3 は，半非同期部分の実装のコード断片である (紙面の都合で大幅に簡略化している)．

- sync_wait と sync_notify は，udos の同期機構であり，モニタの wait/notify，FreeBSD の tsleep/wakeup に相当する．
- FDC_read_write はシステムコールの延長として**同期的**に呼び出される．対照的に FDC_intr_handler はハードウェア割り込みで**非同期的**に呼び出される．

```
FDC_read_write () {
   while (fd_is_busy)
      sync_wait (&fd_is_busy);
   fd_is_busy = 1;
   FDC_seek ();
   sync_wait (&ident_fd);
   FDC_setup_DMAC ();
   read_write_command ();
   sync_wait (&ident_fd);
   fd_is_busy = 0;
   sync_notify (&fd_is_busy);
}
FDC_intr_handler () {
   sync_notify (&ident_fd);
}
```

**図 3 Code fragment for controlling FDD**

複数のプロセスからの並行アクセスを直列化するために，条件変数 fd_is_busy を用いている．また，フロッピーディスクから割り込みが来るまでスリープするために条件変数 ident_fd を用いている．

図 3 が示すとおり，FDC_read_write は sync_wait/sync_notify により抽象化されている．ここで単なる抽象化と異なる点は実装への追跡性[3]が良いことである．例えば，FDD への READ コマンドが 0xE6 で始まる 9 バイトからなることや，sync_wait が「ljmp を使ったタスク切り替えによるプロセススイッチ」で実現されていることはコードをさらに読めばすぐ分かる．しかし，これらは OS の教科書 [14] [15] や OS のソースコード解説本 [9] [10] [11] [12] [13] では分かりにくい．

## 4 議論

我々は教育用 OS では抽象から実装への追跡性が重要であると論じた．これは当然に見えるかも知れないが必ずしもそうではない．例えば，Nachos [4] やハードウェアシミュレータを用いるアプローチ [5] はあまり追跡性を重視していない．Nachos は R3000 をエミュレーションしているが，他のハードウェアを大幅に簡略化してソフトウェアシミュレーションしている．このため，例えば，Nachos のハードディスクは同期的な UNIX ファイルへの読み書きで実現されているため，半非同期部分の実装方法を知るには不適切である．

OS を簡単に作るためであれば上記の追跡性はあまり必要ない．例えば，ブートローダ GRUB [21] や OS 開発キット OSKit [20] [22] を使えばよい．これらはコード

[2] PIT(8254), PIC(8259A), DMAC(8237A), KBDC(8042), VGA(テキスト, 英数字), FDC(82077).

[3] 注意：ここで言う追跡性とは，UML の<<trace>>とは逆向きで，抽象→具象という方向である．

量が多いため追跡性は悪いが，低い実装コストを期待できる．

教育用 OS は特定のハードウェアや実装方法に依存しているが，これは追跡性とのトレードオフである．教育用 OS の目的を考えると，抽象レベルを少し下げて，追跡性を良くするために，ある程度の依存性はやむをえないと考える．

移植性以外にも，追跡性とトレードオフになる性質がいくつもある．例えば，再利用性，拡張性，頑健性，保守性などは 2.1 節で述べたように，間接層を増やすので理解を難しくすることがある．また，多くのデバイスのサポートや高速化も追跡性とトレードオフになりやすい．

## 5 まとめ

本論文では，OS の本質は抽象概念を実現する OS とハードウェアの役割分担にあること，OS 中の抽象概念ではこの役割分担を説明できないこと，udos などの教育用 OS は役割分担の追跡性に優れていることを主張した．

## 参考文献


[1] bochs: the cross platform IA-32 emulator, http://bochs.sourceforge.net/
[2] MINIX information sheet, http://www.cs.vu.nl/~ast/minix.html
[3] PortOS: An Educational Operating System for the Post-PC Environment, B. Atkin, E. G. Sirer, 33rd ACM Tech. Sympo. SIGCSE, pp.116–120, 2002.
[4] W. A. Christopher, S. J. Procter and T. E. Anderson, The Nachos Instructional Operating System, Proc. USENIX Winter, pp.481–488, 1993.
[5] J. Dickinson, Operating systems projects built on a simple hardware simulator, 35rd ACM Tech. Sympo. SIGCSE, pp.320–324, 2000.
[6] D. Hovemeyer, J. Hollingsworth, and B. Bhattacharjee, Running on the Bare Metal with GeekOS, 35rd ACM Tech. Sympo. SIGCSE, pp.315–319, 2004.
[7] Robert Switzer, オペレーティングシステム–設計と実装–, ISBN 4-7819-0725-3, ftp://ftp.gwdg.de/pub/tunix/tunix.tar.Z, 1994.
[8] http://www.tik.ee.ethz.ch/~topsy/
[9] McKusic et al., The Design and Implementation of the 4.4BSD Operating System, Addison Wesley, ISBN 0-201-54979-4, 1996.
[10] A. S. Tanenbaum and A. S. Woodhull, オペレーティングシステム—設計と理論および MINIX による実装 (第 2 版)，プレンティスホール，ISBN 4894710471, 1998.
[11] W. F. Jolitz and L. G. Jolitz, 386BSD カーネルソースコードの秘密，アスキー，ISBN 4756120423, 1998.
[12] J. Lions, Lions' Commentary on UNIX, アスキー, ISBN 4756118445, 1998.
[13] M. Beck et al., Linux カーネルインターナル, ピアソン, ISBN 4756131352, 1999.
[14] A. Silberschatz et al., Applied Operating System Concepts, John Wiley&Sons, ISBN 0471365084, 1999.
[15] A. S. Tanenbaum, Modern Operating Systems, 2nd Ed., Prentice Hall, ISBN 0-13-031358-0, 2001.
[16] H. P. Messmer, The Indispensable PC Hardware Book, 4th Ed. Addison Wesley, ISBN: 0201596164, 2001.
[17] 桑野 雅彦, パソコンのレガシィI/O 活用大全割り込みと DMA からシリアル/パラレル・ポート，FDD/IDE インターフェースまで, ISBN4-7898-3433-6, CQ 出版, 2000.
[18] Intel, インテル・アーキテクチャ・ソフトウェア・ディベロッパーズ・マニュアル, 1999.
[19] Intel, 82077AA CHMOS Single-Chip Floppy Disk Controller, 1994.
[20] The OSKit Project, http://www.cs.utah.edu/flux/oskit/.
[21] GNU GRUB, http://www.gnu.org/software/grub/
[22] B. Ford, K. V. Maren, J. Lepreau, S. Clawson, B. Robinson and J. Turner. The FLUX OS toolkit: Reusable components for OS implementation. proc. 6th IEEE workshop on Hot Topics in Operating Systems, pp.14–19, 1997.
[23] プログラムデザインのためのパターン言語—Pattern Languages of Program Design 選集, PLoPD Editors(著), 細谷 竜一 (翻訳), 中山 裕子 (翻訳), ソフトバンク, ISBN: 4797314397, 2001.
[24] 大場勝，権藤克彦, アスペクト指向を用いたドキュメント整理法の提案, 第 7 回プログラミングおよび応用のシステムに関するワークショップ (SPA2004), 2004
[25] K. Gondow, Homepage for udos, http://www.sde.cs.titech.ac.jp/~gondow/udos/.