OFF2F におけるページ例外回数に着目したページ例外処理時間の評価 – OS カーネル作成処理 –

額田 哲彰[1]　佐藤 将也[1]　谷口 秀夫[1]

概要：バイトアクセス可能な不揮発性メモリが登場し，将来の計算機の実メモリ環境は，揮発性メモリと不揮発性メモリを混載した環境になると考えられる．そこで，この環境を想定した新たな実行ファイル形式として，アクセス形態の違いによりテキスト部とテキスト部以外を 2 つのファイルに分割格納した実行ファイル形式（OFF2F：Object File Format consisting of 2 Files）が提案されている．本稿では，複数のプログラムが繰り返し実行される OS カーネル作成処理を OFF2F プログラムで実行したときの，ページ例外回数に着目したページ例外処理時間の短縮効果を予測する．

1. はじめに

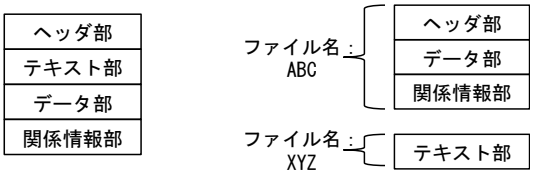
バイトアクセス可能な不揮発性メモリ（Non-Volatile Memory：以降，NVM）が登場している．Intel 社の Optane DC Persistent Memory は，高速で大容量な不揮発性メモリである．しかし，揮発性メモリと比較すると読み込みが低速である [1]．一方で，揮発性メモリと同等の読み込み速度を持つ，高価で小容量な NVM が登場することが予想される．

NVM の登場を受け，NVM を有効利用する研究が行われている．ファイルシステムでの有効利用についての研究として，揮発性メモリと NVM の混載環境における性能向上と一貫性保証を提供するファイルシステム NOVA [2]，ユーザ空間のファイルシステムとカーネル空間のファイルシステムで分割する SplitFS [3]，ユーザレベルのメモリマップド IO を拡張し原子性を提供する Libnvmmio [4]，NVM 上のファイルの読み書きを複写レスで行う SubZero [5] がある．ファイルシステムの他にも，データベースの一種である Key-Value ストアの研究として，NVM をブロックデバイスとして使用することで DRAM の総量を削減する MyNVM [6]，揮発性メモリ，NVM，およびソリッドステートドライブ（以降，SSD）を搭載した環境向けの Key-Value ストアである MatrixKV [7] がある．さらに，NVM のプログラミングをサポートするように Go 言語のコンパイラとランタイムを拡張した go-pmem [8]，永続データへのアクセスからカーネルを排除し，NVM へのシンプルなアクセス環境を提供するデータ中心の OS である Twizzler [9] がある．

揮発性メモリと NVM の混載環境においてプログラム実行を高速化するために，プログラムの実行ファイル形式について NVM を有効利用する研究が行われている．文献 [10] では，揮発性メモリと NVM が混載された計算機において，高速なプログラム実行を可能にする新たな実行ファイル形式（OFF2F：Object File Format consisting of 2 Files）を提案した．この形式では，仮想記憶機構が揮発性メモリと NVM の特徴を生かしたプログラム実行を支援するために，実行ファイルを 2 つのファイルに分割格納している．OFF2F の評価として，ページ例外処理時間の効果予測を行った．具体的には，プログラムのテキスト部とデータ部のサイズ [10]，および Copy on Write（CoW）の有無 [11] に着目して効果予測を行った．また，OS 初期化処理を取り上げ，ページ例外が発生する回数（以降，ページ例外回数）を試算し，ページ例外処理時間を予測した [11]．

本稿では，複数のプログラムが繰り返し実行される処理として OS カーネル作成処理を取り上げ，ページ例外回数を実測し，OFF2F におけるページ例外処理時間の効果予測を行う．また，試算したページ例外回数と実測したページ例外回数の比較，繰り返し実行されるプログラムと一度だけ実行されるプログラムの比較，および，算出したページ例外処理時間と実測したページ例外処理時間の比較により，OS カーネル作成処理におけるページ例外を分析する．

[1] 岡山大学 大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama Univesity



（A）従来の実行ファイル形式　　（B） OFF2F

図 1 実行ファイル形式

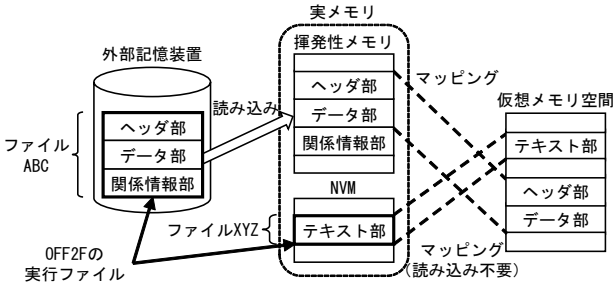


図 2 OFF2F プログラム実行時のマッピング

2. OFF2F とページ例外

2.1 OFF2F

仮想記憶機構において，要求時ページング（ODP：On Demand Paging）機能を利用してプログラムを実行した際，外部記憶装置（例えば磁気ディスク装置（以降，HDD））とメモリ間の入出力時間が長いという問題がある．HDD を利用せず SSD を利用した場合でも，メモリ間複写に比べるとその入出力時間は長い．NVM は読み込みが高速であり，書き込みは低速である．そこで，NVM 上にファイルシステムを構築し，NVM から揮発性メモリへのメモリ間複写を行うことで，ページ読み込み時間を高速化するという方法がある．しかし，実行ファイル全体を NVM 上に格納する必要があるため，大きな NVM が必要となる．したがって，読み出しのみ行われるデータを NVM 上に格納し，仮想記憶を利用してそのまま仮想空間にマッピングできれば，入出力時間を短縮しつつ大きな NVM を必要としない．

実行ファイルは，4 つの内容（テキスト部，ヘッダ部，データ部，および関係情報部）から構成されている．これらはアクセス形態の観点から 2 つに分類できる．ひとつは，読み込みのみ行われるテキスト部，ヘッダ部，および関係情報部である．なお，ヘッダ部の読み出しは，プログラムをプロセスとして実行するときに発生し，このとき，関係情報部の読み出しは発生しない．一方，テキスト部の読み出しは，プログラム実行時に頻繁に発生する．もう一つは，頻繁に読み書きされるデータ部である．

そこで，実行ファイルの内容のアクセス形態に着目した新たな実行ファイル形式（OFF2F：Object File Format consisting of 2 Files）が提案されている．従来の実行ファイルの形式の例と OFF2F の形式を**図 1** に示す．従来の実行ファイル（A）は，各部がすべて同じファイルに格納されている．一方 OFF2F（B）では，実行ファイルが 2 つのファイルに分割格納されている．実行ファイルは 4 つの内容から構成されているため，4 つのファイルに格納できる．しかし，OFF2F では，構造の複雑化を防ぎ，ファイルシステムの占有領域を抑制するため，構成するファイル数を 2 個にしている．

図 1（B）の形式では，ヘッダ部，データ部，および関係情報部をファイル ABC に，テキスト部をファイル XYZ に格納している．プログラム実行時は，ファイル ABC を外部記憶装置に，ファイル XYZ を NVM 上に格納する．プログラム実行時の仮想メモリ空間を構築する処理において，実行ファイルの名前を保持するヘッダ部が外部記憶装置上に存在するため，既存の処理流れを利用できる．また，ファイル XYZ は，必ずしも NVM 上に存在する必要はないため，NVM の有無の影響を受けない．さらに，NVM にはテキスト部のみを格納するため，大きな NVM を必要としない．

2.2 ODP 機能における OFF2F プログラムの実行

仮想記憶機構の ODP 機能において，OFF2F プログラムを実行する場合を以下に述べる．ファイル ABC は外部記憶装置上に存在し，ファイル XYZ は NVM 上に存在する．このときの，仮想メモリ空間へのマッピング処理について，**図 2** に示し，以下に説明する．

(1) 外部記憶装置上に存在するファイル ABC のヘッダ部を読み込む．
(2) ヘッダ部の情報より，テキスト部が NVM 上に存在することを認識し，ページテーブルに NVM のアドレスを登録（マッピング）する．
(3) ヘッダ部の情報より，データ部が外部記憶装置上に存在することを認識し，実メモリを確保する．
(4) 確保した実メモリに外部記憶装置上に存在するデータ部を読み込み，ページテーブルに実メモリのアドレスを登録する．

従来の実行ファイル形式では，テキスト部の外部記憶装置からの入出力時間が必要であった．しかし OFF2F では，NVM 上のテキスト部をそのまま仮想メモリ空間にマッピングできるため，テキスト部の外部記憶装置からの入出力時間を削減できる．

2.3 ページ例外処理

従来の実行ファイル形式に比べ，OFF2F では，テキスト部におけるページ例外処理を簡略化できる．OFF2F プログラムを実行可能な環境における，ODP 機能を有する場合のページ例外処理について，**図 3** に示し，以下に説明する．

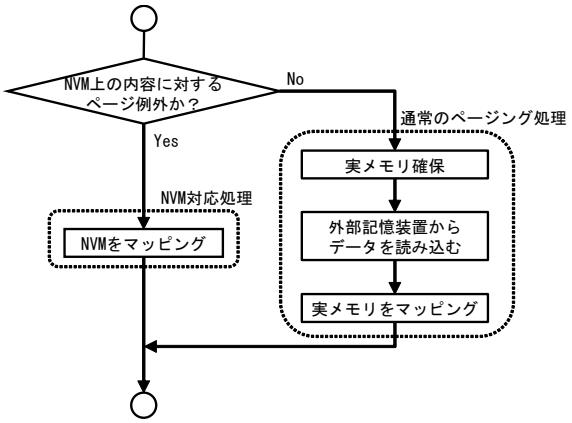


図 3 ページ例外処理の流れ

(1) NVM 上のデータに対する例外でない場合，通常のページング処理を行う．通常のページング処理は，実メモリ確保処理，外部記憶装置からデータを読み込む処理，および実メモリをマッピング表に登録する処理からなる．このとき，実メモリが不足していたら，ページアウト処理を行う．

(2) NVM 上のデータに対する例外の場合，NVM 対応処理として，当該の NVM のページをマッピング表に登録する．

(2) により，NVM 上のテキスト部に対するページ例外の場合ではページインの処理やページアウトの処理で発生する入出力は不要であり，ページイン処理での実メモリの確保処理も省略できる．

3. OS カーネル作成処理

3.1 OS カーネル作成処理におけるプログラム

複数のプログラムが繰り返し実行される処理として，FreeBSD 11.0-RELEASE（以降，FreeBSD）の OS カーネル作成処理を取り上げる．また，文献 [11] で取り上げられた OS 初期化処理と比較することで，その特性を明らかにする．FreeBSD の OS カーネル作成処理で実行されるプログラムのテキスト部とデータ部の大きさ，および各プログラムがリンクするライブラリについて**表 1** に示す．表 1 より，以下のことが分かる．

(1) OS カーネル作成処理において実行されるプログラムは 30 種類であり，プログラムがリンクするライブラリは 16 種類である．

(2) ライブラリをリンクするプログラムは 26 種類であり，as，cc，ld，および make はライブラリをリンクしない．

次に，文献 [11] と同様に，実行プログラムとこれらがリンクするライブラリのページ例外回数を試算する．プロセスは，fork() により複製される．このとき，子プロセスのテキスト部は親プロセスのテキスト部と共有され，データ部は親プロセスと共有されない．つまり，fork() 後の子プロセスで実行するプログラムがデータ部への読み書きを行う際，ページ例外が発生する．また，exec() によりプログラムが起動し，このプログラムがテキスト部の読み込みまたはデータ部への読み書きを行う際，ページ例外が発生する．

したがって，ページ例外回数は，プログラムが fork() する回数や exec() で起動される回数，およびライブラリをリンクするプログラムが fork() する回数や exec() で起動される回数に依存する．具体的には，ページ例外回数は以下で表される．

表 1 OS カーネル作成処理のプログラムとライブラリ

通番	プログラム名	テキスト部のサイズ (Byte)	データ部のサイズ (Byte)	リンクするライブラリ
1	as	1,481,886	20,208	–[*1]
2	awk	192,681	3,528	libm.so.5, libc.so.7
3	cat	8,615	817	libc.so.7
4	cc	43,822,733	17,960	–
5	chmod	4,765	692	libc.so.7
6	cp	15,192	2,080	libc.so.7
7	ctfconvert	98,686	2,736	libdwarf.so.4, libelf.so.2, libz.so.6, libthr.so.3, libc.so.7
8	ctfmerge	67,060	2,320	libelf.so.2, libz.so.6, libthr.so.3, libc.so.7
9	date	16,533	1,248	libc.so.7
10	dirname	2,261	569	libc.so.7
11	echo	2,904	593	libc.so.7
12	find	48,966	1,256	libc.so.7
13	grep	94,331	1,356	libgnuregex.so.5, libbz2.so.4, libbz.so.6, libc.so.7
14	hostname	2,694	593	libc.so.7
15	ld	1,599,486	20,592	–
16	ln	6,736	729	libc.so.7
17	make	704,726	14,896	–
18	mv	10,133	897	libc.so.7
19	nm	91,525	1,129	libdwarf.so.4, libelf.so.2, libc.so.7
20	objcopy	108,814	5,656	libarchive.so.6, libelf.so.2, libc.so.7, libz.so.6, libbz2.so.4, liblzma.so.5, libthr.so.3, libbsdxml.so.4, libcrypto.so.8
21	realpath	2,311	569	libc.so.7
22	rm	10,903	881	libc.so.7
23	sed	32,915	1,108	libc.so.7
24	sh	146,263	1,748	libedit.so.7, libc.so.7, libncursesw.so.8
25	size	15,187	1,036	libelf.so.2, libc.so.7
26	sort	55,312	2,664	libmd.so.6, libc.so.7
27	svnliteversion	1,598,757	3,592	libbsdxml.so.4, libprivatesqlite3.so.0, libz.so.6, libthr.so.3, libc.so.7
28	touch	6,954	705	libc.so.7
29	uudecode	9,218	785	libc.so.7
30	xargs	12,452	939	libc.so.7
	合計	50,270,999	113,882	

[*1] – はリンクするライブラリがないことを示す．

(1) プログラムのテキスト部：ページ数 × exec() で起動される回数
(2) プログラムのデータ部：ページ数 ×（fork() する回数と exec() で起動される回数の和）
(3) ライブラリのテキスト部：ページ数 × リンクするプログラムが exec() で起動される回数
(4) ライブラリのデータ部：ページ数 ×（リンクするプログラムが fork() する回数と exec() で起動される回数の和）

ここで，上記により求まる数値は，そのプログラム全体またはライブラリ全体が一度だけ読まれたときに発生するページ例外回数（以降，全 PF 数）と解釈できる．ただし，exec() により，exec() 前と同じプログラムが起動される場合は，テキスト部を共有する．このため，テキスト部についての exec() で起動される回数には，テキスト部を共有する回数を含まない．

OS カーネル作成処理で実行される各プログラムが fork() する回数，exec() で起動される回数，および全 PF 数を**表 2** に示す．各全 PF 数は，上記 (1)，(2) に基づき算出した．ただし，先に述べたように，exec() によりテキスト部を共有する場合があり，その回数は 4,629 回であった．具体的な内訳は，cc で 4,576 回，sh で 53 回であった．したがって，この場合の回数は全 PF 数の算出に用いない．また，各ライブラリのテキスト部のサイズ，データ部のサイズ，および全 PF 数を**表 3** に示す．各全 PF 数は，表 1 の「リンクするライブラリ」と表 2 の「fork() する回数」や「exec() で起動される回数」を利用して上記 (3)，(4) に基づき算出した．たとえば，libarchive.so.6 の場合，リンクしているプログラムは objcopy のみであり，objcopy の fork() する回数は 0 回であり，exec() で起動される回数は 2,118 回である．したがって，libarchive.so.6 のテキスト部の全 PF 数は $178 \times 2,118 = 377,004$ 回となる．表 2 と表 3 より，プログラムの全 PF 数はテキスト部が 50,089,741 回，データ部が 179,385 回であり，ライブラリの全 PF 数はテキスト部が 12,892,258 回，データ部が 528,207 回である．

3.2 OS 初期化処理との比較

OS カーネル作成処理と OS 初期化処理 [11] を比較する．以下に FreeBSD の初期化処理について示す．

(1) OS 初期化処理において実行されるプログラムは 61 種類であり，プログラムがリンクするライブラリは 37 種類である．
(2) fork() する回数の総数は 580 回であり，exec() で起動される回数の総数は 278 回（うち，テキスト部を共有する場合は 10 回）である．
(3) プログラムの全 PF 数はテキスト部が 2,267 回，データ部が 683 回であり，ライブラリの全 PF 数はテキスト部が 113,433 回，データ部が 9,358 回である．

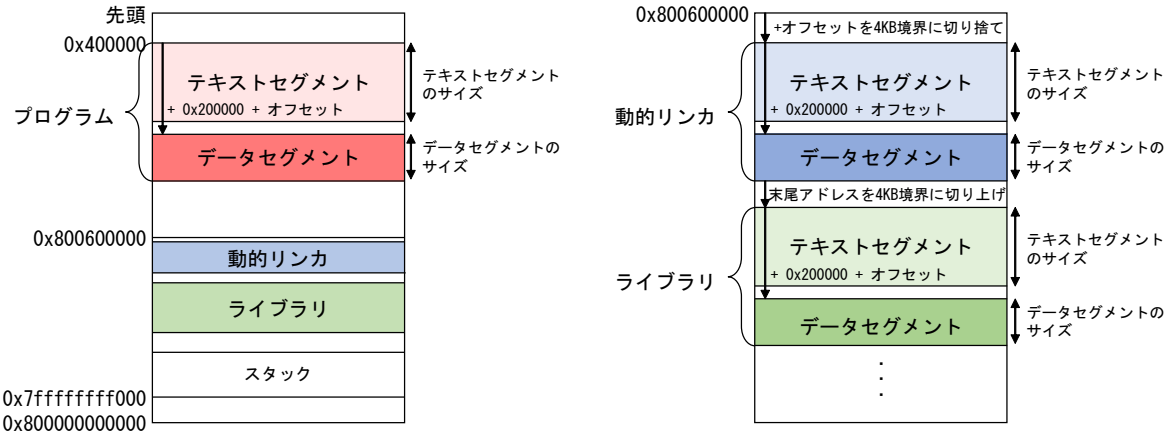
表 2 OS カーネル作成処理におけるプログラムの全 PF 数

通番	プログラム名	fork() する回数	exec() で起動される回数	テキスト部の全 PF 数	データ部の全 PF 数
1	as	0	5	1,810	25
2	awk	858	864	41,472	1,722
3	cat	0	10	30	10
4	cc	4,579	9,133	48,755,343	68,560
5	chmod	0	1	2	1
6	cp	0	1	4	1
7	ctfconvert	0	4,552	113,800	4,552
8	ctfmerge	0	817	13,889	817
9	date	0	1	5	1
10	dirname	0	1	1	1
11	echo	0	128	128	128
12	find	0	3	36	3
13	grep	0	13	312	13
14	hostname	0	1	1	1
15	ld	0	946	369,886	5,676
16	ln	0	5	10	5
17	make	17,785	1,735	300,155	78,080
18	mv	0	44	132	44
19	nm	0	816	18,768	816
20	objcopy	0	2,118	57,186	4,236
21	realpath	0	1	1	1
22	rm	0	7	21	7
23	sed	0	7	63	7
24	sh	1,747	11,504	412,236	13,251
25	size	0	1	4	1
26	sort	0	2	28	2
27	svnliteversion	0	2	782	2
28	touch	0	1	2	1
29	uudecode	0	126	378	126
30	xargs	481	814	3,256	1,295
	合計	25,450	33,659	50,089,741	179,385

表 3 プログラムにリンクされるライブラリの全 PF 数

通番	ライブラリ名	テキスト部のサイズ (Byte)	データ部のサイズ (Byte)	テキスト部の全 PF 数	データ部の全 PF 数
1	libarchive.so.6	728,798	13,416	377,004	8,472
2	libbsdxml.so.4	151,200	7,977	78,440	4,240
3	libbz2.so.4	77,518	4,112	40,489	4,262
4	libc.so.7	1,630,172	48,672	8,671,226	299,112
5	libcrypto.so.8	2,354,308	167,384	1,217,850	86,838
6	libdwarf.so.4	179,422	2,296	236,192	5,368
7	libedit.so.7	210,640	8,143	595,452	26,502
8	libelf.so.2	89,816	2,060	182,688	8,304
9	libgnuregex.so.5	84,145	809	273	13
10	liblzma.so.5	163,044	2,593	84,720	2,118
11	libm.so.5	169,847	1,776	36,288	1,722
12	libmd.so.6	90,148	1,344	46	2
13	libncursesw.so.8	355,179	17,160	996,237	66,255
14	libprivatesqlite3.so.0	1,235,671	15,272	604	8
15	libthr.so.3	107,906	2,964	202,203	7,489
16	libz.so.6	90,999	1,328	172,546	7,502
	合計	7,718,813	297,306	12,892,258	528,207

上記 (1)〜(3)，表 2，および表 3 より，実行されるプログラムとプログラムがリンクするライブラリの種類は，OS 初期化処理の方が多いことが分かる．しかし，fork() する



(A) 仮想メモリ空間全体　　(B) 動的リンカとライブラリの拡大図

図 4　ELF プログラム実行時の仮想メモリ空間

回数や exec() で起動される回数は OS カーネル作成処理の方が多い．このことから，OS カーネル作成処理は，OS 初期化処理よりも複数のプログラムが繰り返し実行される処理である．

4. 評価

4.1 観点

OFF2F におけるページ例外処理回数に着目し，ページ例外処理時間を評価した．具体的には，FreeBSD で使用されている ELF（Executable and Linkable Format）プログラムで OS カーネル作成処理を実行したときのページ例外回数を実測し，OFF2F プログラムで実行したときのページ例外処理時間を予測する．また，ページ例外を以下の観点で分析し，ページ例外処理時間の短縮効果を示す．

(1) ページ例外回数

OS カーネル作成処理を ELF プログラムで実行し，発生するページ例外を実測する．次に，ページ例外が発生した仮想アドレスから，テキスト部で発生したものとテキスト部以外で発生したものに分類する．その後，全 PF 数と，分類したページ例外回数を比較することによって，実際に発生するページ例外の特徴を明らかにする．

(2) 繰り返し実行されるプログラム

OS カーネル作成処理で実行されるプログラムの中から cc を取り上げる．OS カーネル作成処理の中で繰り返し実行される場合と，一度だけ実行した場合で実測したページ例外を比較することで，繰り返し実行されることによるページ例外回数への影響を明らかにする．

(3) ページ例外処理時間

文献 [10] の定式により，(1) において実測したページ例外回数を用いて，ELF プログラムと OFF2F プログラムのページ例外処理時間を算出する．また，OS カーネル作成処理を ELF プログラムで実行しページ例外処理時間を実測する．その後，算出したページ例外処理時間と，実測したページ例外処理時間を比較することによって，テキスト部とテキスト部以外における，それぞれのページ例外処理の内容を明らかにする．

評価環境は以下に示す通りである．

- OS：FreeBSD 11.0-RELEASE
- CPU：Intel Core i3-8100T（3.10 GHz）
- メモリ：8 GB
- HDD：TOSHIBA MQ01ABF050（8 MB Cache）

なお，評価では，OS をシングルユーザモードで起動し，シングルコアの環境で測定を行った．

4.2 ページ例外回数

ELF プログラムを用いて OS カーネル作成処理を実行し，発生したページ例外を実測した．また，実測したページ例外を仮想アドレスによって分類し，テキスト部またはテキスト部以外で発生したページ例外回数を導出した．

ELF プログラム実行時の仮想メモリ空間を**図 4** に示し，以下に説明する．ELF プログラムはプログラムヘッダテーブルと複数のセグメントを持つ．プログラムヘッダテーブルは，各セグメントのオフセットとサイズの情報を含む．また，プログラム実行時は，仮想メモリ空間にテキスト部を含むセグメント（以降，テキストセグメント）と，データ部を含むセグメント（以降，データセグメント）が配置される．また，仮想メモリ空間は，プログラムのセグメント，動的リンカのセグメント，およびライブラリのセグメントに分かれている．仮想メモリ空間上のテキストセグメントの先頭アドレスは，以下で算出される．

(1) 実行プログラム：オフセット＋ 0x400000

(2) 動的リンカ：実行プログラムのデータセグメントのオフセットを 4KB 境界に切り捨てたアドレス＋ 0x800600000

(3) ライブラリ：動的リンカまたはライブラリのデータセグメントの末尾アドレスを 4KB 境界に切り上げたア

ドレス

また，データセグメントの先頭アドレスは，テキストセグメントの先頭アドレス + オフセット + 0x200000 により算出される．

表 1 の各プログラムにおけるテキストセグメントの仮想アドレスを上記 (1)〜(3) により算出した．ページ例外が発生した仮想アドレスがこの仮想アドレスの範囲内にある場合，テキスト部で発生したページ例外とし，そうでない場合はテキスト部以外で発生したページ例外とする．ただし，テキストセグメント内で発生したページ例外はすべてテキスト部で発生したものとする．ページ例外を分類した結果を**表 4** に示す．

たとえば awk の場合，readelf コマンドを使用して，実行プログラムのオフセット，サイズ，およびリンクする動的リンカ（ld-elf.so.1）とライブラリ（libm.so.5 と libc.so.7）の名前を取得した．これにより，各テキストセグメントの仮想アドレスを以下のように算出した．

(1) 実行プログラム：0x400000 – 0x430000
(2) 動的リンカ　　：0x80062f000 – 0x80064e000
(3) libm.so.5　　：0x800850000 – 0x80087a000
(4) libc.so.7　　：0x800a7b000 – 0x800c0a000

表 2，表 3（全 PF 数），および表 4（実測したページ例外回数）より，以下のことが分かる．

(1) テキスト部について，全 PF 数 62,981,999 回（$50,089,741 + 12,892,258$）と比べて実測したページ例外回数は 611 回であり，非常に少ない．つまり，プログラムが繰り返し実行されているが，テキスト部でページ例外はほとんど発生していないことが分かる．

(2) データ部について，全 PF 数 707,592 回（$179,385 + 528,207$）と比べてテキスト部以外の実測したページ例外回数は 22,361,056 回であり，実測したページ例外回数の方が多い．このことから，テキスト部以外で発生したページ例外の多くが，データ部以外の場所でも発生していることが分かる．例えば，スタック，ヒープ，およびコンパイル時の一時ファイルが考えられる．

4.3 繰り返し実行されるプログラム

OS カーネル作成処理で実行されるプログラムの中から cc を取り上げ，4.2 節と同様の方法でページ例外回数を実測した．コンパイル対象は，文字列を画面に出力する簡単なプログラムとした．この結果を**表 5** に示す．

表 4（OS カーネル作成処理）と表 5（cc）より，以下のことが分かる．

(1) cc のテキスト部で発生するページ例外回数について，OS カーネル作成処理では 498 回発生している一方，cc では 408 回発生している．このことから，OS カーネル作成処理における cc のテキスト部で発生するページ例外の多くは，exec() で起動された 9,133 回のうち，最初に起動されたときに発生していると考えられる．

(2) ld のテキスト部で発生するページ例外回数について，OS カーネル作成処理でも cc でも 22 回発生している．このことから，OS カーネル作成処理における ld のテキスト部で発生するページ例外は，exec() で起動された 946 回のうち，最初に起動されたときにほとんど，または全て発生していると考えられる．

(3) プログラムが繰り返し実行される OS カーネル作成処理と 1 度だけ実行される cc を比較した際，テキスト部で発生するページ例外回数に大きな差はない．つまり，プロ

表 4　OS カーネル作成処理のページ例外回数

通番	プログラム名	ページ例外回数	
		テキスト部	テキスト部以外
1	as	20	6,723
2	awk	4	95,148
3	cat	0	635
4	cc	498	15,179,066
5	chmod	0	66
6	cp	0	67
7	ctfconvert	8	3,025,306
8	ctfmerge	0	663,797
9	date	0	74
10	dirname	0	64
11	echo	0	7,934
12	find	0	658
13	grep	7	1,146
14	hostname	0	62
15	ld	22	323,251
16	ln	0	275
17	make	9	440,261
18	mv	0	2,376
19	nm	1	73,523
20	objcopy	20	1,472,684
21	realpath	0	63
22	rm	0	383
23	sed	3	554
24	sh	0	1,002,280
25	size	0	75
26	sort	2	180
27	svnliteversion	17	265
28	touch	0	62
29	uudecode	0	9,187
30	xargs	0	54,891
	合計	611	22,361,056

表 5　cc のページ例外回数

通番	プログラム名	ページ例外回数	
		テキスト部	テキスト部以外
1	cc	408	619
2	ld	22	1,295
3	sh	0	80
	合計	430	1,994

グラムが繰り返し実行された場合も，テキスト部に対するページ例外回数は大きく増加しないと予想できる．

4.4 ページ例外処理時間

文献 [11] では，試算したページ例外回数と文献 [10] で定式化された式を用いて，ページ例外処理時間を算出している．これに倣い，実測したページ例外回数を用いて，OS カーネル作成処理のページ例外処理時間を算出した．

文献 [10] では，図 3 に基づき，ページ例外処理時間を定式化している．具体的には，まず，以下のように各処理時間を定義する．

t_1：実メモリ確保処理時間（1 ページ：4KB）
t_2：外部記憶装置からデータ（4KB）を読み込む時間
t_3：実メモリをマッピング表に登録する処理時間
t_4：NVM をマッピング表に登録する処理時間
P：プログラムにおける（テキスト部＋データ部）が占めるページ数
S：プログラムに占めるテキスト部の割合

次に，プログラムがすべて外部記憶装置上に格納されている場合（ELF）と，テキスト部は NVM 上，他は外部記憶装置上に格納されている場合（OFF2F）におけるページ例外処理時間を，それぞれ以下の式 (1)，(2) のように定式化する．

$$(t_1 + t_2 + t_3)P \tag{1}$$

$$(t_1 + t_2 + t_3)P(1 - S) + t_4 PS \tag{2}$$

なお，上式における $(t_1 + t_2 + t_3)$ は 1 回のページ例外処理時間である．つまり，(1) 式はプログラム全体を，(2) 式はデータ部を必ず外部記憶装置から読み込む際の処理時間である．また，文献 [11] では，P にページ例外回数の総数を，PS にテキスト部のページ例外回数を，$P(1-S)$ にデータ部のページ例外回数を代入することで，ページ例外処理時間を算出している．

ここで，実メモリ確保処理時間（t_1），外部記憶装置からデータ（4KB）を読み込む時間（t_2），および実メモリをマッピング表に登録する処理時間（t_3）を測定した．その結果，$t_1 = 0.168\ \mu$s，$t_2 = 47.6\ \mu$s，および $t_3 = 0.146\ \mu$s であった．ただし，OS カーネル作成処理ではプログラムが繰り返し実行されるため，連続読み込み処理時間の測定結果を t_2 とした．また，NVM をマッピング表に登録する処理は，実メモリでの処理と大きく変わらないと考えられるため，$t_4 = t_3$ と仮定する．式 (1)，(2)，$t_1 \sim t_4$，および表 4 より算出したページ例外処理時間を**表 6** に示す．

また，ELF プログラムにおける OS カーネル作成処理のページ例外処理時間を実測した．4.2 節と同様に，発生したページ例外を分類し，テキスト部で発生したページ例外処理時間と，テキスト部以外で発生したページ例外処理時間を求めた．結果を**表 7** に示す．表 6 と表 7 より，以下のことが分かる．

表 6 （$t_1 = 0.168\ \mu$s，$t_2 = 47.6\ \mu$s，$t_4 = t_3 = 0.146\ \mu$s）として算出したページ例外処理時間 [s]

	テキスト部のページ例外処理時間	テキスト部以外のページ例外処理時間	合計
ELF	2.93×10^{-2}	1,071	1,071
OFF2F	8.92×10^{-5}	1,071	1,071

表 7 テキスト部とテキスト部以外のページ例外処理時間の違いに基づいて算出した OFF2F のページ例外処理時間 [s]

	テキスト部のページ例外処理時間	テキスト部以外のページ例外処理時間	合計
ELF（実測）	3.28	41.1	44.4
OFF2F	8.92×10^{-5}	41.1	41.1

(1) ELF プログラムでのページ例外処理時間を実測したとき，テキスト部で発生したページ例外と，テキスト部以外で発生したページ例外で，1 回あたりのページ例外処理時間が異なる．OS カーネル作成処理で発生するページ例外 1 回あたりの処理時間は，1.99 μs（44.4 s/(611 + 22,361,056)）である．しかし，テキスト部での 1 回あたりのページ例外処理時間は，5.37 ms（3.28 s/611）であり，全体の平均よりかなり大きい（5.37 ms $\gg$ 1.99 μs）．これに対し，テキスト部以外での 1 回あたりのページ例外処理時間は，1.84 μs（41.1 s/22,361,056）である．

(2) 算出したページ例外処理時間と，ELF プログラムで実測したときのページ例外処理時間を比べると，1 回あたりのページ例外処理時間が大きく異なる．テキスト部については，ELF プログラムで実測したときの方が長い（5.37 ms $>$ (0.168 + 47.6 + 0.146) μs）．このことから，外部記憶装置からテキスト部を読み込む際，連続した領域だけでなく非連続な領域も読み込むことがあると予想できる．また，テキスト部以外では，ELF プログラムで実測したときの方が短い（1.84 μs $<$ (0.168 + 47.6 + 0.146) μs）．このことから，文献 [11] に示されている CoW 処理や，実メモリ確保処理と外部記憶装置からデータを読み込む処理を行わずに，マッピング表への登録のみが行われていると予想できる．

4.5 ページ例外処理時間の短縮効果

OFF2F によるページ例外処理時間の短縮効果を考える．4.2 節より，OS カーネル作成処理では，テキスト部に対するページ例外はほとんど発生していない．このため，式 (1)，(2) を用いて，必ず外部記憶装置からデータを読み込むとした表 6 の結果では，ページ例外処理時間の短縮効果は小さい．しかし実際は，4.4 節より，外部記憶装置からのデータの読み込みを伴わないページ例外がテキスト部以

外で多数発生している．

そこで，1 回あたりのページ例外処理時間（$t_1+t_2+t_3$）をテキスト部では 5.37 ms，テキスト部以外では 1.84 μs とする．式 (1)，(2)，($t_1+t_2+t_3$)，t_4，および表 4 より再度算出したページ例外処理時間を表 7 に示す．表 7 より，外部記憶装置からデータの読み込みを伴わない場合を考慮すると，ページ例外処理時間を約 7.41%（$(44.4-41.1)/44.4$）短縮できる．

5. おわりに

FreeBSD の OS カーネル作成処理において，実行ファイル形式を OFF2F で実行することによるページ例外処理時間の短縮効果を示した．

OS カーネル作成処理で発生するページ例外について，ページ例外回数，繰り返し実行されるプログラム，およびページ例外処理時間の観点で分析した．ELF プログラムで実測したページ例外回数について，試算したページ例外回数と比較した場合，テキスト部で発生したページ例外回数は少なく，テキスト部以外で発生したページ例外回数は多いことを明らかにした．また，OS カーネル作成処理で繰り返し実行される cc について，一度だけ実行した場合と比較した結果，テキスト部で発生したページ例外の多くは，プロセスとして最初に起動されたときに発生していることを明らかにした．さらに，ELF プログラムでのページ例外処理時間について，算出した場合と実測した場合を比較した結果，テキスト部とテキスト部以外では 1 回あたりのページ例外処理時間が異なることを明らかにした．この結果に基づいて算出した OFF2F のページ例外処理時間が，OS カーネル作成処理を ELF プログラムで実行したときと比べて，約 7.41 %短縮できることを示した．

謝辞　本研究の一部は，JSPS KAKENHI 18K11244，および共同研究（株式会社富士通研究所）による．

参考文献

[1] Yang, J., Kim, J., Hoseinzadeh, M., Izraelevitz, J. and Swanson, S.: An Empirical Guide to the Behavior and Use of Scalable Persistent Memory, *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pp. 169–182 (2020).

[2] Xu, J. and Swanson, S.: NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories, *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pp. 323–338 (2016).

[3] Kadekodi, R., Lee, S. K., Kashyap, S., Kim, T., Kolli, A. and Chidambaram, V.: SplitFS: Reducing Software Overhead in File Systems for Persistent Memory, *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, p. 494–508 (2019).

[4] Choi, J., Hong, J., Kwon, Y. and Han, H.: Libnvmmio: Reconstructing Software IO Path with Failure-Atomic Memory-Mapped Interface, *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 1–16 (2020).

[5] Kim, J., Soh, Y. J., Izraelevitz, J., Zhao, J. and Swanson, S.: SubZero: Zero-Copy IO for Persistent Main Memory File Systems, *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, p. 1–8 (2020).

[6] Eisenman, A., Gardner, D., AbdelRahman, I., Axboe, J., Dong, S., Hazelwood, K., Petersen, C., Cidon, A. and Katti, S.: Reducing DRAM Footprint with NVM in Facebook, *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18 (2018).

[7] Yao, T., Zhang, Y., Wan, J., Cui, Q., Tang, L., Jiang, H., Xie, C. and He, X.: MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM, *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 17–31 (2020).

[8] George, J. S., Verma, M., Venkatasubramanian, R. and Subrahmanyam, P.: go-pmem: Native Support for Programming Persistent Memory in Go, *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 859–872 (2020).

[9] Bittman, D., Alvaro, P., Mehra, P., Long, D. D. E. and Miller, E. L.: Twizzler: a Data-Centric OS for Non-Volatile Memory, *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 65–80 (2020).

[10] Sato, M. and Taniguchi, H.: OFF2F: A New Object File Format for Virtual Memory Systems to Support Volatile/Non-Volatile Memory-Mixed Environment, *International Journal of Machine Learning and Computing*, Vol. 9, No. 4, pp. 387–392 (2019).

[11] 谷口秀夫，佐藤将也，河辺誠弥，横山和俊：CoW 機能を考慮した OFF2F プログラムのページ例外処理の評価，情報処理学会論文誌，Vol. 61, No. 3, pp. 707–717 (2020).