FUJITSU

FUJITSU Software
Technical Computing Suite V4.0L20

ジョブ運用ソフトウェア
エンドユーザ向けガイド
HPC拡張機能編

J2UL-2535-01Z0(04)
2022年9月

まえがき

本書の目的

本書は、Technical Computing Suite ジョブ運用ソフトウェアに含まれる「HPC(High Performance Computing)拡張機能」のうち、FXサーバを利用するエンドユーザに関係するHPCタグアドレスオーバーライド制御機能およびラージページライブラリについて説明します。

本書の読者

本書は、FXサーバを利用するエンドユーザが対象です。

本書を読むためには、以下の知識が必要です。

- Linux に関する基本的な知識
- 「ジョブ運用ソフトウェア 概説書」による、ジョブ運用ソフトウェアの概要についての知識

本書の構成

本書は、次の構成になっています。

第1章 HPC拡張機能の概要

拡張機能の提供する機能の一覧を説明しています。

第2章 HPCタグアドレスオーバーライド制御機能

HPCタグアドレスオーバーライド制御機能について説明しています。

第3章 ラージページライブラリ

FXサーバにおけるメモリ割り当て機能の概要、ラージページ機能、性能チューニングについて説明しています。

付録A メッセージ

本書の機能が出力するメッセージ集です。

本書の表記について

単位の表現

本書では、単位を表現する際の接頭語は以下のとおりです。基本的にディスクサイズは 10 のべき乗、メモリサイズは2のべき乗で表現します。コマンドの表示や入力時の指定で注意してください。

接頭語	値	接頭語	値
K (kilo)	10^3	Ki (kibi)	2^{10}
M (mega)	10^6	Mi (mebi)	2^{20}
G (giga)	10^9	Gi (gibi)	2^{30}
T (tera)	10^{12}	Ti (tebi)	2^{40}
P (peta)	10^{15}	Pi (pebi)	2^{50}

機種名の表現

本書では富士通製CPU A64FXを搭載した計算機を"FXサーバ"と呼びます。

本書内のアイコンについて

本書では、以下のアイコンを使用しています。

注意

特に注意が必要な事項を説明しています。必ずお読みください。

例

活用例を説明しています。

参照

詳細な情報が書かれている参照先を示しています。

参考

参考情報を説明しています。

輸出管理規制について

本ドキュメントを輸出または第三者へ提供する場合は、お客様が居住する国および米国輸出管理関連法規などの規制をご確認のうえ、必要な手続きをおとりください。

商標

Linux®は米国及びその他の国におけるLinus Torvaldsの登録商標です。

ARM® is the registered trademark of ARM Limited in the EU and other countries.

そのほか、本マニュアルに記載されている会社名および製品名は、それぞれ各社の商標または登録商標です。

出版年月および版数

版数	マニュアルコード
2022年9月 第1.4版	J2UL-2535-01Z0(04)
2021年11月 第1.3版	J2UL-2535-01Z0(03)
2021年8月 第1.2版	J2UL-2535-01Z0(02)
2020年9月 第1.1版	J2UL-2535-01Z0(01)
2020年2月 初版	J2UL-2535-01Z0(00)

著作権表示

変更履歴

変更内容	変更箇所	版数
ラージページライブラリのデフォルト動作に関する注意事項を追加しました。	3.3.1	第1.4版
環境変数XOS_MMM_L_ARENA_LOCK_TYPEの説明を変更しました。	3.4.3	第1.3版
HPCタグアドレスオーバーライド制御機能のメッセージについて記載を変更しました。	A.1	第1.2版
要求されたサイズのラージページ割り当てが失敗したときの、ノーマルページによる割り当て動作の有効化/無効化を選択する設定について記載を追加しました。	3.4.1、A.2	
ラージページライブラリのリンクの順番について記載を追加しました。	3.3.1	第1.1版
キャッシュカラーリングが有効になる条件について説明を改善しました。	3.4.3	

目　次

第1章 HPC拡張機能の概要

本章では、FXサーバ固有の機能をOSやTechnical Computing Suiteで使えるようにサポートする機能(HPC拡張機能)について説明します。

1.1 HPC拡張機能一覧

本節ではHPC拡張機能の提供するエンドユーザ向け機能の一覧を示します。

- HPCタグアドレスオーバーライド制御機能
  アプリケーションのA64FXプロセッサ固有の性能チューニングやプロファイラによるハードウェアイベント情報の取得をサポート。
- ラージページライブラリ
  FXサーバでHuge Page(HugeTLBfs)をより効率よく利用するためのライブラリ。

参照

HPC拡張機能はTechnical Computing Suite ジョブ運用ソフトウェアの提供する各機能と連携して動作します。HPC拡張機能のTechnical Computing Suite ジョブ運用ソフトウェアにおける位置付けについては、マニュアル「ジョブ運用ソフトウェア 概説書」を参照してください。

第2章 HPCタグアドレスオーバーライド制御機能

本章では、HPC拡張機能のうち、HPCタグアドレスオーバーライド制御機能について説明します。

2.1 HPCタグアドレスオーバーライド制御機能

本節では、A64FXプロセッサ固有のHPCタグアドレスオーバーライド機能の説明と、それを制御するユーザコマンドについて説明します。

2.1.1 HPCタグアドレスオーバーライド機能

HPCタグアドレスオーバーライド機能は、FXサーバの計算ノードに使われるA64FXプロセッサ固有の機能で、性能フラグを制御するものです。

この性能フラグの制御は、A64FXプロセッサのARMv8-A Address tagging機能に関連した設定をすることで実現しています。ARMv8-A Address tagging機能は仮想アドレスの上位8ビットをハードウェア(MMU:メモリ管理ユニット)が解釈しないようにするARMv8プロセッサの機能です。

FXサーバでは、固有機能であるHPCタグアドレスオーバーライド機能の方を有効にし、ARMv8-A Address tagging機能の方は無効にして運用します。

このため、FXサーバ利用者が、ARMv8-A Address tagging機能を使用するつもりでアプリケーションプログラムを作成し、ジョブ実行すると、意図しない性能関連フラグが設定され性能低下など意図しない動作となる可能性があります。

そこで、ARMv8-A Address tagging機能を使いたいユーザのために、HPCタグアドレスオーバーライド機能を制御するユーザコマンド(HPCタグアドレスオーバーライド制御機能)を提供します。

2.1.2 fhetboコマンド

fhetboコマンドは、HPCタグアドレスオーバーライド制御機能として、HPCタグアドレスオーバーライド機能の有効、無効を切り替えます。

HPCタグアドレスオーバーライド機能を無効にすると、ARMv8-A Address tagging機能を使えます。

【名前】

fhetbo - HPCタグアドレスオーバーライド機能を制御します

【書式】

/opt/FJSVxos/fhehpc/bin/fhetbo {enable|disable}

【オプション】

enable: HPCタグアドレスオーバーライド機能を有効にします。

disable: HPCタグアドレスオーバーライド機能を無効にします。

【説明】

ジョブに割り当たっているすべてのコアに対し、HPCタグアドレスオーバーライド機能を有効化または無効化します。

オプションの指定は必須で、無指定のときにはコマンドの使用方法を表示します。

本コマンドはジョブの中でだけ使えます。

利用シーンとしては、ジョブスクリプトの中で本コマンドを呼び出すことを想定します。

本コマンドの実行はノード単位で行われ、コマンドが効果を及ぼす範囲は、コマンドが実行されたノード内の、同一ジョブに割り当てられたコアに限定されます。よってマルチノードジョブで利用する場合には、mpiexecコマンド経由で実行する必要があります。

ジョブの中での本コマンドの呼び出し回数や状態の上書きに特に制限はありません。最後に実行したコマンドの指示が有効となり、enable/disableを繰り返し変更できます。ジョブ終了後には、HPCタグアドレスオーバーライド機能は有効に戻ります。

HPCタグアドレスオーバーライド機能を有効/無効にした結果はメッセージとして出力されます。

詳細は"付録A メッセージ"をご確認ください。

例

ジョブスクリプトの作成例を記します。

```
#!/bin/bash
#PJM -L "node=256"
#PJM -L "elapse=86400"

export PATH=<directory>:$PATH
export NR_PROCS=256

mpiexec -n $NR_PROCS /opt/FJSVxos/fhehpc/bin/fhetbo disable   ← HPCタグアドレスオーバーライド機能を無効化
mpiexec -n $NR_PROCS ./a.out                                  ← プログラム a.out を実行
mpiexec -n $NR_PROCS /opt/FJSVxos/fhehpc/bin/fhetbo enable    ← HPCタグアドレスオーバーライド機能を有効化
```

参照

Technical Computing Suite ジョブ運用ソフトウェアのジョブ実行環境およびジョブスクリプトの記述方法については、マニュアル「ジョブ運用ソフトウェア エンドユーザ向けガイド」を参照してください。

mpiexecコマンドについては、Technical Computing Suite Development Studioの マニュアル「MPI使用手引書」を参照してください。

第3章 ラージページライブラリ

本章では、HPC拡張機能のうち、ラージページライブラリについて説明します。

HPC分野における大規模なメモリを使用するアプリケーションプログラムでは、OSがメモリを管理するコストがアプリケーションプログラムの実行性能に影響を与えます。このコストを減らすために、HPC拡張機能では独自に拡張したラージページ機能を提供しています。

ラージページライブラリの理解に必要な基本的なメモリ割り当ての考え方や、FXサーバにおけるメモリ構成(ジョブ用メモリ分割機能)などについても説明します。

3.1 メモリ割り当て機能の概要

本節では、FXサーバの拡張機能としてのメモリ割り当て機能の概要として、以下の機能と処理を説明します。

・ラージページ機能

・ページング方式

3.1.1 ラージページ機能

ラージページ機能は、Linuxの標準機能であるHugeTLBfsを拡張し、大規模なデータを扱うアプリケーションプログラムに対して、通常のページ(ノーマルページ)より大きなページサイズのメモリ(ラージページ)を割り当てることで、OSのアドレス変換処理によるコストを低減し、メモリアクセス性能を向上させる機能です。

3.1.1.1 メモリアドレス変換とTLB

アプリケーションプログラムがメモリアクセスをする際には、仮想メモリアドレスを使います。そのため、仮想メモリアドレスから物理メモリアドレスへ変換する必要があります。このアドレス変換には、主記憶上に存在するアドレス変換テーブルを使用します。その際、さらに高速アクセスするため、CPU内に存在するロード・ストア機構がTLB(Translation Look-aside Buffer)というアドレス変換バッファを使用して物理アドレスを求めます。

CPUのロード・ストア機構は、動作中のアプリケーションプログラムからメモリへのロード・ストア要求を受け取ります。ロード・ストア機構によるメモリアドレス変換の概要を図3.1 仮想メモリアドレスから物理メモリアドレスへの変換に示します。

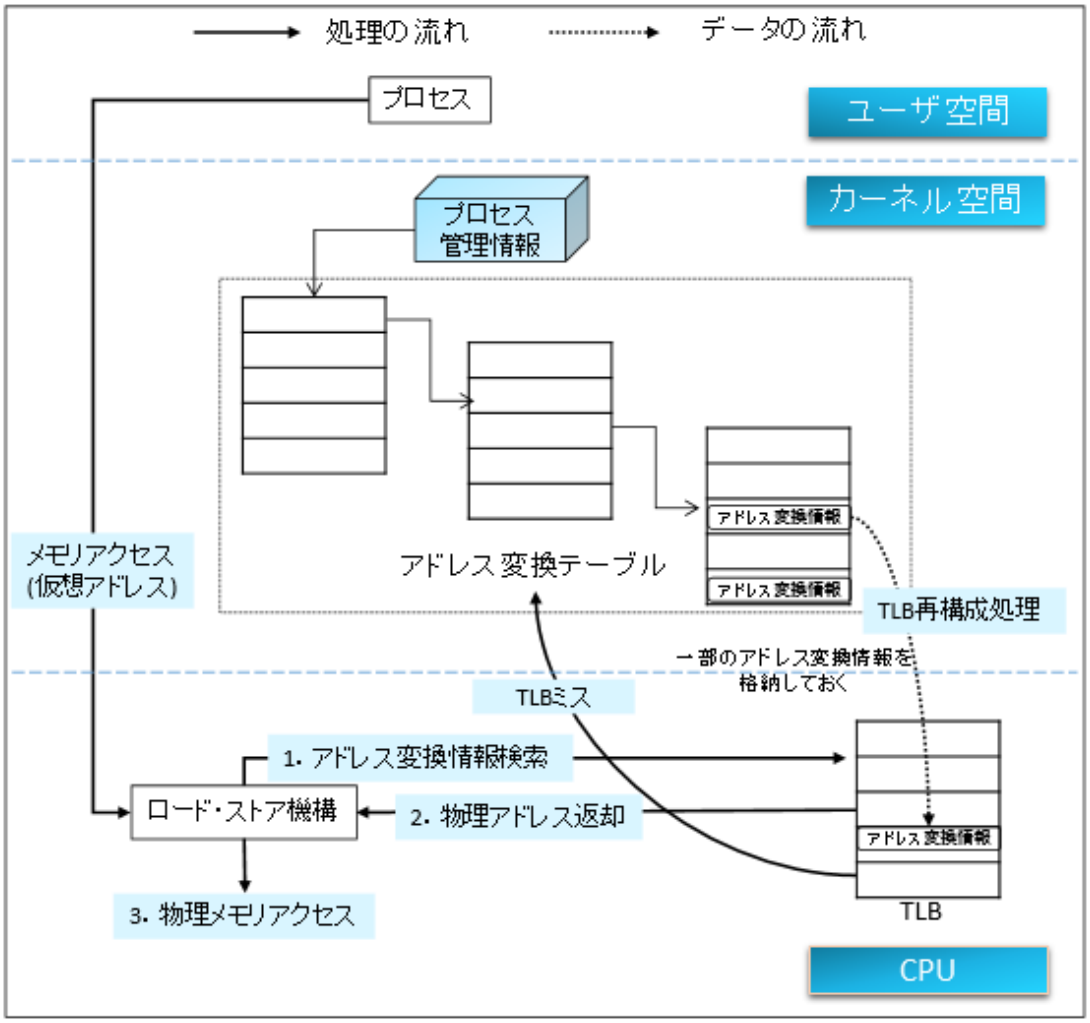
1. ロード・ストア先として指定された仮想メモリアドレスに対応する物理メモリアドレスをTLBから求めます。

   TLBに対応するアドレス変換情報がなければTLBミスを発生させ、アドレス変換テーブルからアドレス変換情報を求め、TLBへ読み込みます。

2. TLBから該当するアドレス変換情報により物理メモリアドレスを入手します。
3. ロード・ストア先に対して、物理メモリアクセスを開始します。

TLBから物理メモリアドレスが求められなかったときには、TLBミスが発生します。TLBミスが発生した場合にはTLB再構成処理(アドレス変換テーブルの再読み込み)が必要となり、一般的にその処理には大きなコスト(時間)がかかります。

図3.1 仮想メモリアドレスから物理メモリアドレスへの変換



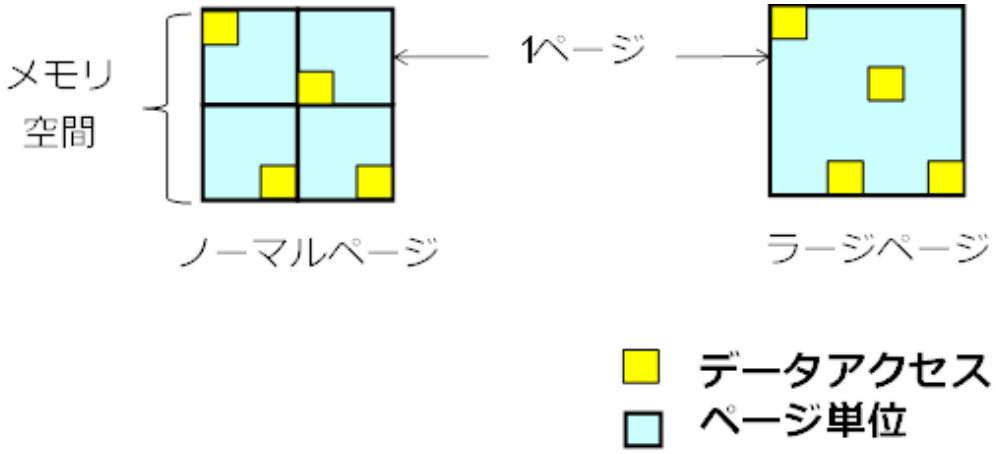
3.1.1.2 ノーマルページ・ラージページ

FXサーバのOSではページサイズとして、ノーマルページで64KiB、ラージページで2MiBを利用可能です。ラージページ機能ではデフォルトでラージページ化が有効となっています。"3.4 ラージページライブラリ設定用環境変数"の節に記載した環境変数(XOS_MMM_L_HPAGE_TYPE)によってラージページの有効化/無効化を選択できます。

大規模なメモリを扱うアプリケーションプログラムにおいては、ノーマルページではアクセスするページ数が多くなり、TLB再構成処理の頻度が高くなる傾向があります。ラージページを使用した場合、ノーマルページと比較して、アクセスするページが少なくなるため、TLB再構成処理の頻度は低くなります。

つまり、大量のメモリを使用するアプリケーションプログラムに対して、ラージページを割り当ててTLBミスを削減することで、OSのTLB再構成処理によるコストを低減し、メモリアクセス性能を向上させることができます。

図3.2 ノーマルページとラージページのデータアクセスイメージ



アプリケーションプログラムが初めてアクセスするメモリアドレスに対しては、アドレス変換情報がTLBに存在しないため、TLBミスが発生します。図3.2 ノーマルページとラージページのデータアクセスイメージを例に取ると、ノーマルページの場合、あるメモリ空間に対し4つのページへのデータアクセスにより4回のTLBミスがするのに対して、ラージページの場合は、同じメモリ空間に対して同じページ内のデータアクセスとなるため、TLBミスが1回だけで済みます。

3.1.1.3 ページサイズの違いによるメリット・デメリット

ページサイズに 64KiB、2MiB を利用した際のそれぞれのメリット・デメリットを以下に示します。

表3.1 ページサイズによるメリット・デメリット

評価項目	64KiB	2MiB
TLBミス率	高い	低い
メモリ初期化コスト	小さい	大きい
メモリ使用効率	高い	低い

1. ページサイズ 64KiBは、ノーマルページです。TLBミス率は高いものの、メモリ初期化コストが小さく、メモリ使用効率が高いため、メモリ使用量が少ないアプリケーションプログラムに対しては有効な場合があります。
2. ページサイズ 2MiBは、ラージページです。64KiBよりメモリ初期化コストが大きくなり、メモリ使用効率が低くなりますが、TLBミス率が低くなります。そのため、FXサーバのラージページ機能では、ラージページ(2MiB)利用をデフォルトとしています。

アプリケーションプログラムは、典型的な通信で使用する通信バッファなどの小規模メモリから計算用の大規模メモリまで多彩なサイズのメモリを使用します。

アプリケーションプログラムのメモリ使用量が少ないときにラージページを利用すると、メモリ使用効率が悪くなる可能性があります。例えば、ヒープ領域のメモリ消費量が 1MiB のみの場合でも、ラージページを利用した場合、2MiB のメモリが確保されることになるため、利用されないメモリ領域が 1MiB 発生します。

逆に、アプリケーションプログラムのメモリ使用量が多いときにノーマルページを使用すると、TLB ミス率が高くなり、実行性能が向上しない可能性が高くなります。

3.1.2 ページング方式

ページング方式には、デマンドページング方式とプリページング方式の2種類があります。FXサーバのラージページ機能では、"3.4 ラージページライブラリ設定用環境変数"の節に記載した環境変数(XOS_MMM_L_PAGING_POLICY)によって、静的データ(.bss領域)、スタック/スレッドスタック領域、および動的メモリ確保領域の各メモリ領域にページング方式を設定できます。

1. デマンドページング方式とは、アプリケーションプログラム実行中に必要なページが主記憶に存在しない場合に、必要に応じてページを主記憶に割り当てる方式のことです。最初にメモリ領域にアクセスしたタイミングで物理ページを割り当てます。
2. プリページング方式とは、あらかじめ主記憶にページを割り当てておく方式のことです。メモリ領域を割り当てたタイミングで物理ページを割り当てます。

図3.3 ページング方式の違いによるメモリ割り当ての違いで、簡単なNUMA構成を例にとり、ページング方式の違いによるアプリケーションプログラムの動作の違いを示します。以下の例では、1つのCPU(32cores)に2つのNUMAが存在するものとし、それぞれNUMA#0, NUMA#1と呼びます。

なお、FXサーバの実際のNUMA構成の詳細については、"3.2.2 ジョブ用メモリ分割機能"を参照ください。

図3.3 ページング方式の違いによるメモリ割り当ての違い

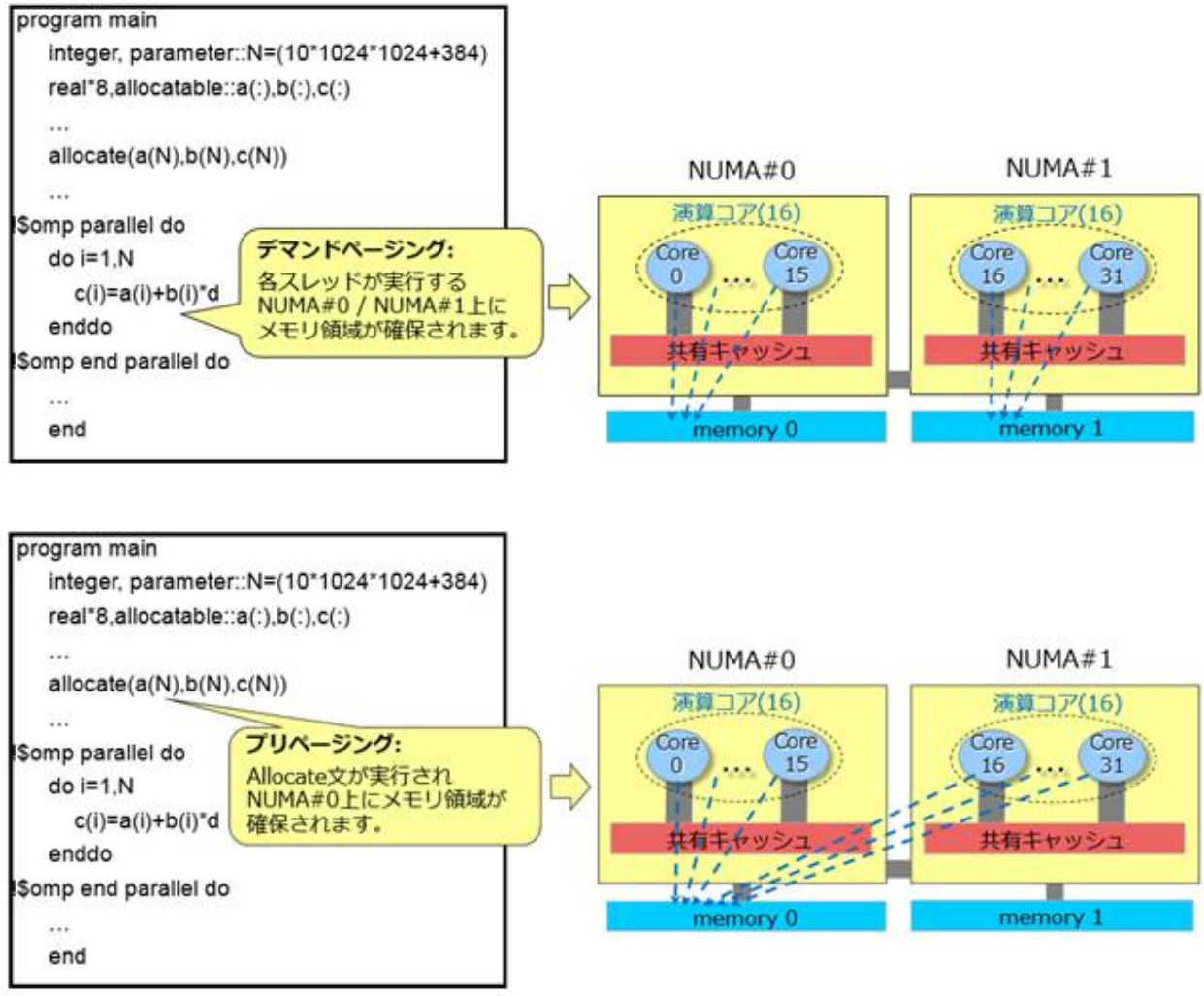


表3.2 ページング方式の違いによるメモリアクセスコスト

ページング方式	NUMA内/外のメモリアクセス	初回メモリアクセス
デマンドページング方式	できる限りNUMA内のメモリを使用 (コスト:低い)	メモリアクセス時にページを物理メモリにロード (コスト:高い)
プリページング方式	NUMA内外のメモリに関係なく使用 (コスト:高い)	ページをあらかじめ物理メモリにロード (コスト:低い)

表3.2 ページング方式の違いによるメモリアクセスコストに示すように、メモリアクセス性能に影響する要因は2つあります。1つ目は、ある演算コアからアクセスする物理メモリが、同じNUMA内にあるか否かです。演算コアと同じNUMAに属する物理メモリへのアクセスよりも、異なる物理メモリへのアクセスは処理コストがかかります。2つ目は、初回メモリアクセス時のコストです。デマンドページング方式を選択した場合、メモリアクセス時に物理メモリを割り当てるため、初回メモリアクセス時にページロード処理コストがかかります。

例えば、4コアのスレッド並列アプリケーションプログラムで個々のスレッドで動的にメモリ領域を獲得する場合には、プリページング方式からデマンドページング方式にすると、演算コアと同じNUMA内の物理メモリアクセスの頻度が増え、メモリアクセス性能の向上が期待できます。

3.2 FXサーバのメモリ構成とメモリ割り当て

FXサーバでは、ジョブ用に必要な量のラージページを確保するため、ジョブ用メモリ分割機能によって、NUMAノードを仮想的にシステム用とジョブ用に分割します。本節は、ジョブ用メモリ分割機能を説明します。

3.2.1 システム用メモリとジョブ用メモリ

各計算ノードでは、アプリケーションプログラムが使用するメモリ(ジョブ用メモリ)とシステムが使用するメモリ(システム用メモリ)の使用量をそれぞれ分けて集計しています。ラージページが使用されるのは、ジョブ用メモリのみとなります。

- システム用メモリ

  IO処理用ページ(ページキャッシュ)やOS本体が動作する際に使用されるメモリです。割り当てられるページサイズは64KiB、割り当て方式はデマンドページング方式です。

- ジョブ用メモリ

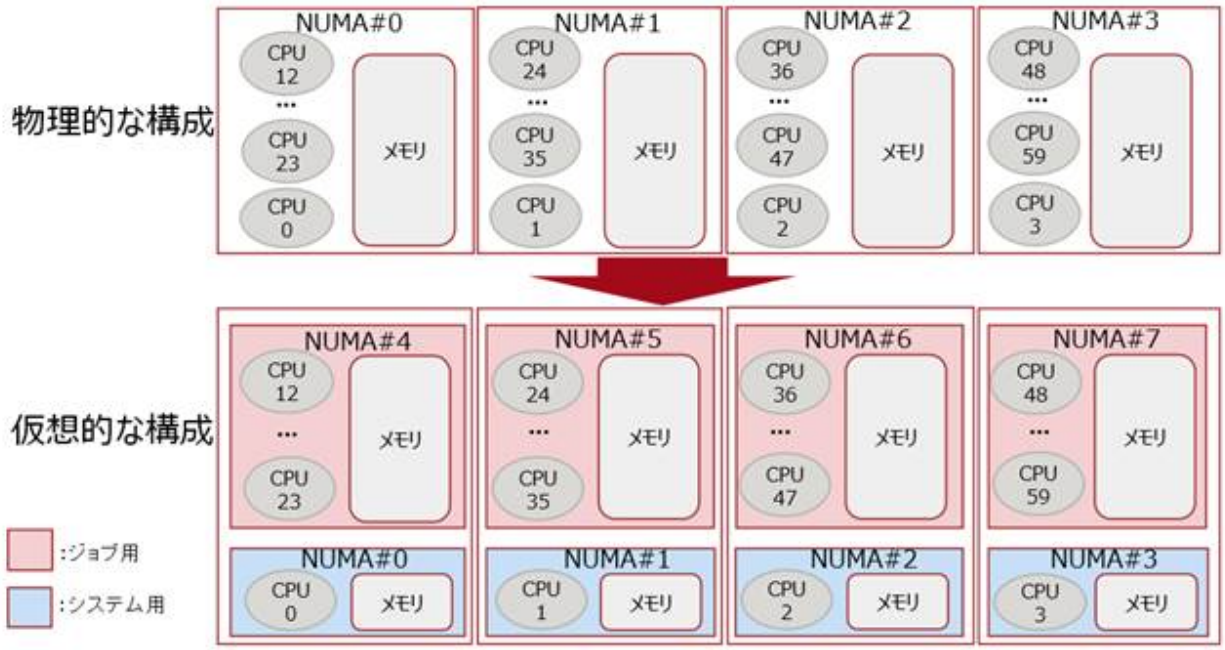
  アプリケーションプログラムが使用するデータに割り当てられるメモリです。FXサーバでは2MiBサイズのラージページを使用します。ページング方式は、エンドユーザが"3.4 ラージページライブラリ設定用環境変数"の節に記載した環境変数を使って指定できます。

3.2.2 ジョブ用メモリ分割機能

FXサーバは、物理的には4つのNUMAノード構成ですが、各物理NUMAノードはさらに仮想的にシステム用、ジョブ用の2つのNUMAノード構成に分割しています。(NUMA#0～#3をシステム用メモリ、NUMA#4～#7をジョブ用メモリに割り当てます。)

これにより、エンドユーザが実行するジョブの処理で行われるメモリアクセスが、システムの処理で行われるメモリアクセスに影響を受けないように分離され、ジョブの実行性能の向上が見込めます。

図3.4 FXサーバのNUMAノード構成



注意

エンドユーザがジョブとして実行するアプリケーションプログラムが使用するCPUは、ジョブ用メモリを割り当てたNUMA#4～NUMA#7にあるCPU12～CPU59です。(システムが使用するCPUはCPU0～CPU3です。CPU4～CPU11は欠番です。)

マルチスレッドのアプリケーションプログラムで、sched_getaffinity(2)で取得したCPU番号を使ってプロセス(スレッド)を割り当てるような処理についてはそのまま正常に動作しますが、CPU番号について、構成を意識しない静的な値で処理をする場合には、CPU番号が12から始まることを考慮してプログラミングする必要があります。

CPU番号をFXサーバの構成を意識してプログラミングする例については、"3.4.3 チューニング用の設定"のサンプルプログラム("環境変数XOS_MMM_L_ARENA_LOCK_TYPEを使ったチューニングによる効果の確認例")を参照してください。

3.3 FXサーバのラージページ

FXサーバの拡張機能としてラージページライブラリ(libmpg.so)を提供します。本節では、FXサーバのラージページライブラリを利用する際の、各領域のページサイズおよびアプリケーションプログラムのデータの配置について説明します。

3.3.1 FXサーバのメモリ領域とラージページ化対象

FXサーバのラージページライブラリは、領域のうち、静的データ(.bss領域、.data領域)、動的メモリ確保領域(mmap領域)、スレッドヒープ領域、スタック領域、およびスレッドスタック領域をラージページ(2MiBページ)化します。テキスト領域、動的メモリ確保領域(ヒープ領域)、および共有メモリ領域についてはラージページ化の対象外です。

表3.3 FXサーバのメモリ領域とラージページ化対象

領域	ラージページ化対象	領域の使用用途
テキスト(.text)領域	×(64KiBページ)	アプリケーションプログラム(a.out)の命令列が配置されるメモリ領域
静的データ(.data)領域	○(2MiBページ)	アプリケーションプログラム(a.out)の静的データが格納されるメモリ領域(初期化有り)
静的データ(.bss)領域	○(2MiBページ)	アプリケーションプログラム(a.out)の静的データが格納されるメモリ領域(初期化無し)
動的メモリ確保領域(ヒープ領域)	×(64KiBページ)	プロセスヒープ領域/メインスレッド用ヒープ領域 ラージページライブラリでは、本メモリ領域はラージページ化されません。 メモリ領域は、以下の時に割り当てられます。 brk(2)/sbrk(2)システムコールで獲得するか、または、ラージページライブラリをリンクしない状態で、環境変数MALLOC_MMAP_THRESHOLD_で指定したサイズより小さいサイズの動的メモリ確保要求(malloc(3))を発行する時に割り当てられる。
スレッドヒープ領域	○(2MiBページ)	サブスレッド用のヒープ領域
スタック領域	○(2MiBページ)	プロセススタック領域/メインスレッド用スタック領域 (ラージページ化には環境変数XOS_MMM_L_LPG_MODEによる明示指定が必要)
スレッドスタック領域	○(2MiBページ)	サブスレッド用のスタック領域 (ラージページ化には環境変数XOS_MMM_L_LPG_MODEによる明示指定が必要)
動的メモリ確保領域(mmap領域)	○(2MiBページ)	mmap(2)を発行する時に割り当てられるメモリ領域 メモリ領域は、以下の時に割り当てられます。 ラージページライブラリをリンクして動的メモリ確保要求(malloc(3))を発行する時に割り当てられる。 または、ラージページライブラリをリンクしない状態で、環境変数MALLOC_MMAP_THRESHOLD_で指定したサイズより大きいサイズの動的メモリ確保要求(malloc(3))を発行する時に割り当てられる、または、スレッドヒープ領域/スレッドスタック領域としても割り当てられる。
共有メモリ	×(64KiBページ)	プロセス間でのメモリ共有のために使用

ラージページライブラリlibmpg.soはFXサーバ向けに最適化されたラージページ機能で、HugeTLBfsのオーバーコミット機能を利用することで、事前のメモリプールを予約せず、ユーザI/Fの使い勝手を改善します。Development Studioのコンパイラで -Klargepage オプション指定することで、アプリケーションプログラムが本ライブラリを利用するようになります。libmpg.soを使わずLinux標準のラージページ機能を利用する場合、Development Studioのコンパイラで -Knolargepage オプションを指定します。ノーマルページを利用する場合には、環境変数XOS_MMM_L_HPAGE_TYPE=noneを明示的に指定してください。

本項で説明した環境変数の詳細については"3.4 ラージページライブラリ設定用環境変数"の節をご覧ください。

参照

ノーマルページまたはラージページを利用するプログラムをDevelopment Studioで翻訳する方法(-K{largepage|nolargepage})については、Development Studioのマニュアル「Fortran使用手引書」、「C言語使用手引書」および「C++言語使用手引書」を参照してください。

注意

- ラージページライブラリのリンクの順序について

  ラージページライブラリの指定(Development Studioのコンパイラで指定する-lmpgオプション)は、エンドユーザが明示的に指定するライブラリ(例:-lcや-lpthread)より常に前に来るよう指定してください。この順序を守らない場合、正常にラージページが割り当てられない場合があります。

- brk(2)/sbrk(2)で獲得されたメモリ領域について

  カーネルの仕様によりbrk(2)/sbrk(2)で獲得されたメモリ領域はラージページ化されません。そのため、ラージページライブラリlibmpg.soをリンクした状態で、brk(2)/sbrk(2)とmalloc(3)(環境変数MALLOC_MMAP_THRESHOLD_より大きいサイズ要求時)を混在利用すると、ノーマルページとラージページが混在することになり、メモリ獲得の量とタイミングによっては、メモリが獲得できずに、brk(2)/sbrk(2)がENOMEMで終了することがあります。

  ラージページライブラリlibmpg.soをリンクする場合は、malloc(3)を利用することを推奨します。

  ラージページライブラリlibmpg.soをリンクしたアプリケーションプログラムでbrk(2)/sbrk(2)を利用したい場合は、環境変数XOS_MMM_L_HPAGE_TYPE=noneを明示的に指定してください。

- 頻繁にシグナルを発行するような処理を行うアプリケーションについて

  ラージページライブラリ(libmpg)をリンクし、プリページング方式を設定した状態で、timer_create(2)システムコールを使用し、頻繁にSIGALRM/SIGVTALRMなどの任意のシグナルを発生させるような処理を行うと、アプリケーションから発行されたfork(2)(clone(2))がシグナルを受けて、システムコールがERESTARTNOINTRで復帰し続けます。その結果、fork(2)(clone(2))の再実行が頻発し、終了しない現象が発生することがあります。

  このような場合は、ページング方式をデマンドページング方式に設定することで現象を回避できます。

  環境変数XOS_MMM_L_PAGING_POLICY=demand:demand:demandを明示的に指定してアプリケーションを実行してください。

- ラージページライブラリのデフォルト動作について

  ラージページライブラリのデフォルト動作では、アプリケーションプログラム (a.out) の起動時に一時的に「静的データ (.data) 領域」と「静的データ (.bss) 領域」を合わせたバイトサイズ(※1)の 2 倍のメモリを使用します。
  そのため、大きな静的データ領域を持つアプリケーションプログラムの場合、起動時にメモリ不足のため、プログラムが SIGKILL で強制終了することがあります。
  ※1 サイズは readelf -S コマンドなどで確認できます。この 2 倍のサイズのメモリを一時的に使用します。

  この場合、ラージページライブラリのデフォルト動作を変更することで、回避できる可能性があります。
  静的データ (.bss) 領域のページング方式のデフォルトをデマンドページングにすることで、プログラム起動時のメモリ使用量を抑えられます。

  以下に、pjsub コマンドの -x オプションで環境変数 XOS_MMM_L_PAGING_POLICY を設定し、静的データ (.bss) 領域のページング方式を変更する例を示します。

```
$ pjsub -x XOS_MMM_L_PAGING_POLICY=demand:demand:prepage jobscript.sh
```

  環境変数 XOS_MMM_L_PAGING_POLICY の説明は「3.4.2 ページング方式の設定」を参照してください。

参考

一般的なコンパイラ(gccなど)でも、本ラージページライブラリlibmpg.soをリンクすることでラージページ化したアプリケーションプログラムを作成できます。ほかのライブラリ(下記例では"-lc -lpthread")を明示的にリンクする場合、ラージページライブラリはそれらのライブラリより常に前に来るよう指定してください。

ラージページ化するにあたり、本ラージページライブラリは、以下のパスでラージページ用のリンカスクリプトも提供しています。これは、静的データ(.data)、および静的データ(.bss)をラージページ化するためのものです。このリンカスクリプトもコンパイラに適切に指定してください。

/opt/FJSVxos/mmm/util/bss-2mb.lds

以下にgccでコンパイルする場合の例を示します。

```
例) gcc -Wl,-T/opt/FJSVxos/mmm/util/bss-2mb.lds -L/opt/FJSVxos/mmm/lib64 -lmpg -lc -lpthread test_program.c
```

なお、アプリケーションプログラムがPIE(Position Independent Executable、位置独立実行形式)でコンパイルされている場合、.data/.bss領域はラージページ化されません。この場合、ラージページライブラリは警告ログを出力するだけで、.data/.bss領域にはノーマルページが使用され、アプリケーションプログラムは実行を継続します。例えばgccで明示的にPIE形式とならないようにコンパイルするには、-no-pie オプションを使用してください。

3.3.2 アプリケーションプログラムのデータが配置されるメモリ領域

アプリケーションプログラム内の変数は、その定義の仕方によって配置されるメモリ領域が異なります。そのため、各メモリ領域のページサイズを指定する場合は、変数がどのメモリ領域に配置されるのかを意識する必要があります。

本項では、Fortran、C、C++のソースコードを例に、変数の宣言の仕方でどのメモリ領域に配置されるかを説明します。各領域に対するページサイズは、"3.3.1 FXサーバのメモリ領域とラージページ化対象"の"表3.3 FXサーバのメモリ領域とラージページ化対象"のように割り当てられます。

- Fortran

```
program main
integer*8, parameter::N=(1024_8)
real*8 a(N)                    !a は初期値なしローカル配列
real*8 :: b(N)=1.0             !b は初期値あり配列
real*8,allocatable::c(:)       !c は割付配列
allocate(c(N))
...
end
```

ローカル配列 a は静的データ領域(.bss)に配置されます。ただし、-Kautoまたは-Kthreadsafeが有効なオプションで翻訳したアプリケーションプログラムの場合にはプロセススタック領域に配置されます。配列 b は静的データ領域(.data)に配置されます。配列 c は動的メモリ確保領域(ヒープ領域またはmmap領域)に配置されます。

- C言語

```
#define N 1024
double a[N];                          //a は初期値なしグローバル変数
double b[N]={1.0};                    //b は初期値ありグローバル変数
double *c;
double *d;                            //c,d はポインタ変数
void *func_thread(void *args) {
  double f[N];                        //f はローカル変数
  d=(double *)malloc(sizeof(double)*N);
  ...
}
int main(void){
  double e[N];                        //e はローカル変数
  c=(double *)malloc(sizeof(double)*N);
  ...
```

```
  pthread_t pthread;
  pthread_create( &pthread, NULL, &func_thread, (void *)NULL);
  …
}
```

グローバル変数 a は静的データ領域(.bss)に配置されます。グローバル変数 b は静的データ領域(.data)に配置されます。ポインタ変数 c,d は動的メモリ確保領域(ヒープ領域またはmmap領域)に配置されます。ローカル変数 e はプロセススタック領域に配置されます。ローカル変数 f はスレッドスタック領域に配置されます。

注意

各プログラミング言語でスレッド並列アプリケーションプログラムを実行した場合に、各スレッドのスタック領域として専用のスレッドスタック領域を用意します。

- C++

```
const int N = 1024;
struct Klass {
  double k;
  Klass() : k (0.0) {}
  Klass(double K) : k (K) {}
};
std::vector<Klass> a(N);          //vectorクラスによる領域確保
int main(){
    Klass* b = new Klass[N];      //new演算子による領域確保
    std::vector<double> c(N);     //vectorクラスによる領域確保
    return 0;
}
```

変数 a, b, c はすべて動的メモリ確保領域(ヒープ領域またはmmap領域)に配置されます。

注意

C++の変数に割り当てられるメモリ領域はC言語と同じですが、vectorクラスなどにより動的に獲得されるメモリ領域は動的メモリ確保領域(ヒープ領域またはmmap領域)として割り当てられます。

3.4 ラージページライブラリ設定用環境変数

ラージページライブラリの動作を調整するために、本節で示す環境変数が利用できます。

3.4.1 ラージページライブラリの基本設定

本項ではラージページライブラリの基本的な設定をするための環境変数を示します。

表3.4 ラージページライブラリ基本設定用環境変数

変数名	指定値	デフォルト値	詳細
XOS_MMM_L_HPAGE_TYPE	hugetlbfs|none	hugetlbfs	ラージページライブラリによるラージページ割り当て動作の有効化/無効化を選択する設定です。 「hugetlbfs」の場合、HugeTLBfsによるラージページ化をします。 「none」の場合、ラージページライブラリによるラージページ化は行いません。この指定の場合、"XOS_MMM_L_" で始まるラージーライブラリの環境変数の指定はすべて無効です。

変数名	指定値	デフォルト値	詳細
			指定値以外の値を指定した場合は、「hugetlbfs」を指定したものとみなします。 ラージページ化については注意事項があります。表の下部にある注意("ラージページ化したときの /proc/*pid*/maps のスタック領域の表示について")を参照してください。 [McKernelモードだけ] 「none」の場合、McKernelの拡張THPモードを使用してラージページ化します。
XOS_MMM_L_LPG_MODE	base+stack|base	base+stack	スタック領域およびスレッドスタック領域のラージページ割り当て動作の有効化/無効化を選択する設定です。 「base+stack」の場合、静的データおよび動的メモリ確保領域だけではなく、スタック領域およびスレッドスタック領域もラージページ化します。 「base」の場合、静的データおよび動的メモリ確保領域のみラージページ化します。スタック領域およびスレッドスタック領域はラージページ化しません。 指定値以外の値を指定した場合は、「base +stack」を指定したものとみなします。 スタック領域のラージページ化については注意事項があります。表の下部にある注意("スタック領域のラージページ化に伴うアライメントについて")を参照してください。
XOS_MMM_L_HUGETLB_FALLBACK	0|1	0	要求されたサイズのラージページ割り当てが失敗したときの、ノーマルページによる割り当て動作の有効化/無効化を選択する設定です。 「0」の場合、要求されたサイズのラージページ割り当てに失敗したとき、OOM(Out of memory) killerによりプロセスがkillされます。 「1」の場合、獲得可能なサイズでラージページを割り当て、残りの分にはノーマルページを割り当てます。ノーマルページの割り当てにも失敗したときには、OOM killerによりプロセスがkillされます。 本環境変数については注意事項があります。表の下部にある注意 ("XOS_MMM_L_HUGETLB_FALLBACK 有効化時の動作について" および " XOS_MMM_L_HUGETLB_FALLBACKのジョブ統計情報について")を参照してください。
XOS_MMM_L_PRINT_ENV	on|off|1|0	0	アプリケーションプログラムのデバッグ向けの設定です。 本環境変数に「1」または「on」を指定した場合、ラージページライブラリが提供する性能チューニング用環境変数の一覧を標準エラー出力に出力します。出力は、アプリケーションプログラムの開始時(main関数の実行より前)に1度だけ行われます。「0」または「off」を指定した場合、性能

変数名	指定値	デフォルト値	詳細
			チューニング用環境変数の一覧を標準エラー出力に出力しません。 指定値以外の値を指定した場合、「0」を指定したとみなします。 なお、main関数の実行後、mallopt(3)によって変更される設定は出力に反映されません。

McKernelモードでは、上記のほかに下記の環境変数が使えます。

表3.5 ラージページライブラリ基本設定用環境変数[McKernelモードだけ]

変数名	指定値	デフォルト値	詳細
XOS_MMM_L_HUGETLB_SZ	2M|32M	2M	割り当てるラージページの大きさを選択する設定です。(「2M」または「32M」と指定すると、それぞれ2MiBページ、32MiBページが設定されます。) McKernelがサポートしている場合、指定した大きさでのラージページ割り当てをします。 指定値以外の値を指定した場合は、「2M」を指定したものとみなします。

参照

McKernelモードなどのジョブ実行環境については、マニュアル「ジョブ運用ソフトウェア エンドユーザ向けガイド」の"ジョブ実行環境"を参照してください。

注意

- ラージページ化したときの /proc/*pid*/maps のスタック領域の表示について(*pid*:プロセスID)

  ラージページ化にあたり、/proc/*pid*/maps の表示内容が一部書き換わります。以下に例を示します。(/proc/*pid*/numa_maps も同様です。)

  ー 静的データ(.bss領域/.data領域)、動的メモリ確保領域(ヒープ領域/mmap領域)

  (ラージページ化する前)

  ```
  00430000-00890000 rw-p 00000000 00:00 0                                  [heap]
  ```

  (ラージページ化した後)

  ```
  aaaae2400000-aaaae2600000 rw-p 00000000 00:0e 452989                     /anon_hugepage (deleted)
  ```

  ー スタック領域(プロセススタック領域/メインスレッド用スタック領域)

  (ラージページ化する前)

  ```
  fffffffd0000-1000000000000 rw-p 00000000 00:00 0            [stack]
  ```

  (ラージページ化した後)

  ```
  fffaeb800000-1000000000000 rw-p 00000000 00:0e 274404  /memfd: [stack] by libmpg (deleted)
  ```

・ スタック領域のラージページ化に伴うアライメントについて

ラージページ化したスタック領域のサイズは、基本的に ulimit -s に設定されたsoft limit値を元に、割り当てるHugeTLBfsページのサイズにアラインします。

また、割り当てるHugeTLBfsページの開始・終了アドレスも同様にアラインアップまたはダウンします。

マップするアドレスとサイズをアラインした結果、スタック領域が隣接する領域とオーバーラップする可能性があります。オーバーラップした場合、ラージページ化は行いません。警告メッセージを出力し、通常ページのままアプリケーションプログラムを続行します。

・ XOS_MMM_L_HUGETLB_FALLBACK 有効化時の動作について

XOS_MMM_L_HUGETLB_FALLBACK 有効化時(「1」を指定時)に、ノーマルページの割り当ての対象となるメモリ領域は、動的メモリ確保要求(malloc(3))を発行するときに割り当てられる動的メモリ確保領域(mmap領域)です。

XOS_MMM_L_HUGETLB_FALLBACK に有効化を指定しても、下記のAND条件を満たさない場合、機能が無効化されます。下記は、ラージページライブラリの環境変数をデフォルト値で使用すれば、条件を満たします。

1. XOS_MMM_L_HPAGE_TYPE=hugetlbfs かつ、
2. XOS_MMM_L_PAGING_POLICY=*:*:prepage かつ、
3. XOS_MMM_L_ARENA_LOCK_TYPE=1 かつ、
4. XOS_MMM_L_MAX_ARENA_NUM=1

・ XOS_MMM_L_HUGETLB_FALLBACK のジョブ統計情報について

XOS_MMM_L_HUGETLB_FALLBACK が有効に機能したかどうかの情報をノード毎のジョブ統計情報に追加できます。追加する場合、ジョブ運用ソフトウェアのジョブ統計情報の設定を管理者に依頼してください。

本項目の出力には、papjmstats.conf ファイル(システム管理ノード: /etc/opt/FJSVtcs/papjmstats.conf)の Item サブセクションの項目名を以下のように定義してください。

－ ItemName : hugetlb_fallback

－ ItemNameDisp : HUGETLB_FALLBACK_LPG (※変更可能)

－ RecordNameList : JN

－ DataType : PJMX_DATATYPE_UINT8

－ DispFormat : dec

papjmstats.conf ファイルの設定例を以下に記します。

```
Cluster {
...
    Item {
        ItemName=hugetlb_fallback
        ItemNameDisp=HUGETLB_FALLBACK_LPG
        RecordNameList=JN
        DataType=PJMX_DATATYPE_UINT8
        DispFormat=dec
    }
...
    File {
        ...
        JN {
            ITEM=hugetlb_fallback
        }
        ...
    }
...
}
```

上記を設定すると、.statsファイル(pjsub コマンドの -s/-S オプション指定時に出力されるファイル)に下記の行が表示されます。(「0」: 無効時、「1」: 有効時)

```
HUGETLB_FALLBACK_LPG        : 1
```

ジョブ統計情報の設定については、ジョブ運用ソフトウェアのマニュアル「管理者向けガイド ジョブ管理編」の"ジョブ運用管理機能の設定"を参照してください。

3.4.2 ページング方式の設定

本項ではラージページライブラリのページング方式を設定するための環境変数を示します。

表3.6 ページング方式設定用環境変数

変数名	指定値	デフォルト値	詳細
XOS_MMM_L_PAGING_POLICY	[demand|prepage]:[demand|prepage]:[demand|prepage]	prepage:demand:prepage	各メモリ領域のページング方式(ページの割り当て契機)を選択する設定です。 「demand」はデマンドページング方式、「prepage」はプリページング方式を意味します。 本変数はコロン(:)区切りで3つのメモリ領域のページング方式を指定します。 第1指定は、静的データの.bss領域です。(静的データの.data領域はページング方式指定の対象外で常にprepageとなります。) 第2指定は、スタック領域およびスレッドスタック領域です。 第3指定は、動的メモリ確保領域です。 指定値以外の値を指定した場合は、「prepage:demand:prepage」を指定したものとみなします。 プリページング方式では、その後のアプリケーションプログラムのページフォールトの発生を抑えられ、性能改善および性能ブレが小さくなる傾向があります。ただし、アプリケーションプログラムが自前でプリページング(例えばゼロクリアなど)を実装している場合や、メモリの一部の領域にだけアクセスする場合などには、プリページング方式を選択しないほうが良いケースもあります。

3.4.3 チューニング用の設定

本項ではラージページ割り当てに関するチューニングをするための環境変数を示します。

これらの環境変数のうち、"XOS_MMM_L_"で始まるものは、ラージページライブラリで独自に追加したものです。そのほかの環境変数はglibcの変数です。

表3.7 チューニング用環境変数

変数名	指定値	デフォルト値	詳細
XOS_MMM_L_ARENA_FREE	1|2	1	free(3)で解放されるヒープ領域の扱いに関する設定です。 「1」を指定した場合は、解放可能なメモリを即時に解放します。「2」を指定した場合は、メモリを一切解放せず、全メモリをプールして再利用します。「1」、「2」以外の値を指定した場合、「1」を指定したものとみなします。

変数名	指定値	デフォルト値	詳細
			「2」の場合、MALLOC_MMAP_THRESHOLD_で指定した値以上のサイズのメモリ要求もヒープ領域から割り当て、解放後も空きメモリ領域として保持し続けます。ヒープ領域の解放処理は行いません。メモリ使用効率低下防止のため単一のヒープ領域ですべてのサイズのメモリ割り当てをする必要があるため、スレッドヒープ領域は生成しません。すなわち「2」は以下の設定の組み合わせと等価です。 XOS_MMM_L_ARENA_LOCK_TYPE=1 XOS_MMM_L_MAX_ARENA_NUM=1 MALLOC_MMAP_THRESHOLD_=ULONG_MAX MALLOC_TRIM_THRESHOLD_=ULONG_MAX
XOS_MMM_L_ARENA_LOCK_TYPE	0|1	1	メモリ割り当てポリシーに関する設定です。「0」はメモリ獲得性能優先、「1」はメモリ使用効率優先を意味します。「0」、「1」以外の値を指定した場合、「1」を指定したものとみなします。 「0」の場合、メインアリーナが競合すると、新たにスレッドヒープ領域を生成します。「1」の場合、メインアリーナが競合すると、最大アリーナ数(XOS_MMM_L_MAX_ARENA_NUM)以内であればスレッドごとにスレッドヒープ領域を生成し、そうでなければロックが獲得できるまで生成を待ちます。 マルチスレッドアプリケーションプログラムでメモリ獲得要求を同時に呼び出した場合、「0」は並列処理が可能ですが、メモリ使用効率が低下します。「1」はメモリ獲得が逐次処理となりますが、メモリ使用効率が向上します。 マルチスレッドアプリケーションプログラムで各スレッドのメモリ獲得要求が競合し、かつ、競合が頻発するようなケースでは、設定を「0」にすることで性能が改善する可能性があります。
XOS_MMM_L_MAX_ARENA_NUM	1以上INT_MAX以下の整数値 [10進数]	1	XOS_MMM_L_ARENA_LOCK_TYPE=1のときのみ有効な変数で、生成可能なアリーナ(プロセスヒープとスレッドヒープ領域の総和)の数を設定できます。生成するスレッドヒープ領域の数を制限したい場合に使用してください。 デフォルト設定(「1」)の場合、プロセスヒープ領域のみを使用し、スレッドヒープ領域は生成されません。この場合、XOS_MMM_L_ARENA_LOCK_TYPE=1と等価です。設定をn(≧2)とした場合、プロセスヒープ領域のほか、(n-1)個のスレッドヒープ領域が生成される可能があります。
XOS_MMM_L_HEAP_SIZE_MB	MALLOC_MMAP_THRESHOLD_の2倍以上	MALLOC_MMAP_THRESHOLD_の2倍	スレッドヒープ領域を使用する場合に、スレッドヒープ領域の生成時および拡張時に獲得するメモリサイズを設定します。

変数名	指定値	デフォルト値	詳細
	ULONG_MAX以下の整数値<MiB単位> [10進数]		デフォルト値はMALLOC_MMAP_THRESHOLD_の2倍で、スレッドのメモリ獲得の初回時にデフォルト値のスレッドヒープ領域が生成されます。1スレッド当たりの総獲得メモリ量がデフォルト値を超えるとさらに指定値のスレッドヒープ領域を生成(拡張)します。 各スレッドが獲得するメモリ量が少ないと予想される場合、本環境変数の値を小さく設定することでメモリ使用効率が向上する可能性があります。
XOS_MMM_L_COLORING	0|1	1	キャッシュカラーリングの有無の設定です。 キャッシュカラーリングをすると、プロセッサのL1キャッシュのコンフリクトを軽減します。 「0」の場合、キャッシュカラーリングを行いません。 「1」の場合、MALLOC_MMAP_THRESHOLD_(デフォルトは128MiB)以上のサイズで行われるmmap(2)によるメモリ獲得時にはキャッシュカラーリングをします。MALLOC_MMAP_THRESHOLD_未満のサイズで行われるヒープ領域からのメモリ獲得時にはキャッシュカラーリングを行いません。 キャッシュカラーリングが必ず有効になる条件は下記です。 1) MALLOC_MMAP_THRESHOLD_ 以上のサイズのmalloc(3)要求をする、かつ、 2) XOS_MMM_L_FORCE_MMAP_THRESHOLD=1を指定する。(これにより必ずmmap(2)によりメモリ領域を割り当てます。) 「0」、「1」以外の値を指定した場合は、「1」を指定したものとみなします。 一般的にはキャッシュカラーリングをおこなったほうが性能を改善する傾向にありますが、アプリケーションプログラムが自前でカラーリングを実装している場合は、本ライブラリのキャッシュカラーリングを無効にしたほうが良いといえます。
XOS_MMM_L_FORCE_MMAP_THRESHOLD	0|1	0	MALLOC_MMAP_THRESHOLD_(デフォルトは128MiB)以上のサイズのメモリ獲得時にmmap(2)を優先するかどうかの設定です。 「0」の場合、mmap(2) は優先しません。まずヒープ領域の空きを検索し、空きがあればヒープ領域の空きメモリを返します。ヒープ領域の空きが見つからないときにのみ mmap(2) でメモリを獲得します。 「1」の場合、mmap(2) を優先します。ヒープ領域の空きは検索せず、(例え空きがあっても) mmap(2) でメモリを獲得します。 「0」、「1」以外の値を指定した場合、「0」を指定したものとみなします。

変数名	指定値	デフォルト値	詳細
MALLOC_CHECK_	0|1|2|3|5|7 [10進数]	3	プログラミングエラー(メモリ破壊や二重解放など)の検出に関する設定です。設定に従い、エラー検出時に以下のアクションをします。 0 : 無視して処理を継続する。 1 : 詳細なエラーメッセージを出力し、処理を継続する。 2 : アプリケーションプログラムをabortする。 3 : 詳細なエラーメッセージ、スタックトレース、メモリマッピングを出力し、アプリケーションプログラムをabortする。 5 : 簡単なエラーメッセージを出力し、処理を継続する。 7 : 簡単なエラーメッセージ、スタックトレース、メモリマッピングを出力し、アプリケーションプログラムをabortする。 これらの値以外の値を指定すると、2進数で下位3ビットの値に対応した動作をします。 なお、すべてのエラーを検出できるわけではなく、メモリリークなどは検出できません。
MALLOC_TOP_PAD_	0以上ULONG_MAX以下の整数値<byte単位> [10進数]	131072 (=128KiB)	ヒープ領域を伸長する際の1回当たりの伸長サイズを設定します。ページサイズで切り上げた値が使用されます。 アプリケーションプログラムが一度に獲得・解放するメモリ量より大きい値を設定することで、システムコールの発行回数が軽減し、メモリ獲得性能が向上することがあります (ただし、メモリ使用効率は低下する可能性があります)。
MALLOC_PERTURB_	INT_MIN以上INT_MAX以下の整数値 [10進数]	0	アプリケーションプログラムのデバッグ向けの設定です。メモリ獲得(calloc(3)を除く)およびメモリ解放時に本環境変数に指定された値に基づいてメモリ領域を埋めます。メモリ獲得時は指定された値の最下位バイトの補数、メモリ解放時には最下位バイトを書き込みます。 メモリ領域を初期化せずに使用している、または、解放済みのメモリ領域を参照しているといった問題を検出する際に使用します。
MALLOC_MMAP_MAX_	INT_MIN以上INT_MAX以下の整数値 [10進数]	2097152 (=2*1024*1024)	MALLOC_MMAP_THRESHOLD_(デフォルトは128MiB)以上のサイズのメモリ獲得時にmmap(2)でメモリ獲得する回数の上限値の設定です。 mmap(2)によるメモリ獲得時に現在値を1カウントアップします。free(3)で解放すると現在値を1カウントダウンします。 上限値はプロセス単位で設定されます。 「0」に設定すると、メモリ獲得サイズやMALLOC_MMAP_THRESHOLD_の設定値に関係なくmmap(2)ではなくヒープ領域からメモリを獲得します。

変数名	指定値	デフォルト値	詳細
MALLOC_MMAP_THRESHOLD_	0以上ULONG_MAX以下の整数値<byte単位> [10進数または16進数]	134217728 (=128MiB)	本環境変数で指定したサイズ以上のサイズのメモリ獲得要求は、mmap(2)によってメモリを獲得します。本環境変数で指定したサイズ未満のメモリ獲得要求はヒープ領域からメモリを獲得します。 mmap(2)で獲得したメモリをfree(3)すると、メモリは即時に解放されます。一方、ヒープ領域から獲得したメモリをfree(3)すると、ヒープ領域のトップにMALLOC_TRIM_THRESHOLD_以上の連続した空き領域ができない限り、メモリを即時に解放せずプールします。プールしたメモリは再利用が可能です。 本設定値を大きくするとヒープ領域で管理する最大メモリサイズが増大し、メモリ領域の再利用が促進されメモリ獲得性能が向上することがあります(ただし、メモリ使用効率は低下する可能性があります)。
MALLOC_TRIM_THRESHOLD_	0以上ULONG_MAX以下の整数値<byte単位> [10進数または16進数]	134217728 (=128MiB)	ヒープ領域から獲得したメモリをfree(3)する場合に、メモリを即時に解放するかどうかの閾値の設定です。指定した閾値以上の連続した空きメモリがヒープ領域のトップにできる場合、メモリを即時に解放します。連続した空きメモリが閾値未満の場合、メモリを解放せずにプールします。 本設定値を大きくすると一度獲得したメモリを解放する頻度が低下し、メモリ領域の再利用が促進されメモリ獲得性能が向上することがあります(ただし、メモリ使用効率は低下する可能性があります)。

例

環境変数XOS_MMM_L_ARENA_FREEを使ったチューニングによる効果の確認例

- **アプリケーションプログラムの説明**

  1. 8MiBサイズのヒープ領域のメモリを1024回獲得(malloc(3))して、解放(free(3))します。
  2. 1の処理を2回ループ実行します。
  3. 各回のmalloc(3)/free(3)の時間を測定します。

- **アプリケーションプログラム**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <time.h>

#define N 1024
#define MALLOC_CNT 1024
#define DATA_CNT (1024*1024*1024)

clock_t time_start;
clock_t time_end;
double *c[MALLOC_CNT];  //heap memory
double a[DATA_CNT];  //data memory
```

```
int main(int argc, char *argv[]){
    int i;
    long sec;
    long nsec;
    int loop=0;
    struct timespec time1 ={0,0};
    struct timespec time2 ={0,0};

    while(loop <2){
        printf("malloc start.¥n");
        clock_gettime(CLOCK_REALTIME, &time1);
        for(i=0;i<MALLOC_CNT;i++){
            c[i]=(double *)malloc(sizeof(double)*N*N);
            if (c[i] == NULL) {
                fprintf(stderr, "malloc error: cnt=%d, errno=%d¥n", i, errno);
                exit(1);
            }
        }
        clock_gettime(CLOCK_REALTIME, &time2);
        printf("malloc end.¥n");
        sec = (time2.tv_sec - time1.tv_sec);
        nsec= (time2.tv_nsec-time1.tv_nsec);
        if(nsec<0){
            sec--;
            nsec += 1000000000L;
        }
        printf("MALLOC TIME:%d:%010d¥n", sec, nsec);
        sleep(10);

        printf("free start.¥n");
        clock_gettime(CLOCK_REALTIME, &time1);
        for(i=0;i<MALLOC_CNT;i++){
            free(c[i]);
        }
        clock_gettime(CLOCK_REALTIME, &time2);
        printf("free end.¥n");
        sec = (time2.tv_sec - time1.tv_sec);
        nsec= (time2.tv_nsec-time1.tv_nsec);
        if(nsec<0){
            sec--;
            nsec += 1000000000L;
        }
        printf("FREE TIME:%d:%010d¥n", sec, nsec);
        loop++;
    }
    return EXIT_SUCCESS;
}
```

・ **チューニング方法**

環境変数 XOS_MMM_L_ARENA_FREE を以下の2つのパターンで設定し、性能を比較します。

1. free(3)時に解放可能なメモリページを即時解放するように設定

   export XOS_MMM_L_ARENA_FREE=1

2. free(3)時に解放可能なメモリページを即時解放しないように設定

   export XOS_MMM_L_ARENA_FREE=2

・ **解説・性能予測**

XOS_MMM_L_ARENA_FREE=1を設定した場合、1回目のfree(3)を実行後にmmappedチャンクに確保されたメモリ領域は即時解放します。その後、2回目のmalloc(3)処理では、mmappedチャンクにもう一度メモリ領域を確保するために1回目のmalloc(3)処理とほぼ同じ時間がかかると考えられます。

XOS_MMM_L_ARENA_FREE=2を設定した場合、1回目のfree(3)を実行後にヒープ領域に確保されたメモリ領域は解放されません。2回目のmalloc(3)処理では、ヒープ領域に確保されているメモリ領域を再利用するため、メモリ割り当て処理コストが削減され、malloc(3)処理時間が短くなると考えられます。

例

環境変数XOS_MMM_L_ARENA_LOCK_TYPEを使ったチューニングによる効果の確認例

- **アプリケーションプログラムの説明**

  1. 複数のスレッドを作成し、各スレッドを別々のCPUに割り当てます。
     なお、このプログラムでは、割り当てるCPU番号を、FXサーバの構成を意識して明示的に指定しています。FXサーバでアプリケーションプログラムが使用するCPU番号については、"3.2.2 ジョブ用メモリ分割機能"を参照してください。
  2. 各スレッドはそれぞれmalloc(3)を発行しており、ヒープ領域を確保します。
  3. 1,2の内容をループ10回で実行し、毎回のmalloc(3)/free(3)の時間を測定します。

- **アプリケーションプログラム**

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <sched.h>
#include <errno.h>
#include <sys/time.h>

#define KBYTES (1024)
#define CPU_NUM (16)
#define MAX_MALLOC_CNT (5*64*KBYTES)

pthread_t thread[CPU_NUM];
pthread_barrier_t barrier;
int malloc_size = 64*KBYTES;
int malloc_cnt = MAX_MALLOC_CNT/CPU_NUM;
int loop_cnt = 10;
char *strp[MAX_MALLOC_CNT];
int cpuid[CPU_NUM]={};
int cpunum[CPU_NUM]={};

void* thread_main(void *arg) {
    int cpuid = *(int*)(arg);
    int i;
    pthread_barrier_wait(&barrier);
    for (i=0;i<malloc_cnt;i++) {
        strp[i+cpuid*malloc_cnt] = (char*)malloc(malloc_size);
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    int i, ret, loop;
    cpu_set_t cpu;
    struct timeval st, et, rt;

    if ( 2 == argc ) {
        malloc_size = atoi(argv[1]);
    } else if ( 3 == argc ) {
        malloc_size = atoi(argv[1]);
        malloc_cnt = atoi(argv[2]);
```

```
    } else if ( 4 == argc ) {
        malloc_size = atoi(argv[1]);
        malloc_cnt = atoi(argv[2]);
        loop_cnt = atoi(argv[3]);
    }

    loop = 0;
    while ( loop < loop_cnt ) {
        ret = pthread_barrier_init(&barrier, NULL, CPU_NUM+1);
        for (i=0;i<CPU_NUM;i++) {
            /* Since the core number usable by the user starts from number 12, 12 is added to i */
            cpuid[i] = i+12;
            cpunum[i] = i;
            ret = pthread_create(&thread[i], NULL, thread_main, (void*)(&cpunum[i]));
            if ( 0 != ret ) {
                fprintf(stderr, "pthread_create: errno=%d¥n", ret);
                exit(1);
            }

            CPU_ZERO(&cpu);
            CPU_SET(cpuid[i], &cpu);
            ret = pthread_setaffinity_np(thread[i], sizeof(cpu_set_t), &cpu);
            if ( 0 != ret ) {
                fprintf(stderr, "pthread_setaffinity_np: errno=%d¥n", ret);
                exit(1);
            }
        }

        pthread_barrier_wait(&barrier);
        gettimeofday(&st, NULL);
        for (i=0;i<CPU_NUM;i++) {
            ret = pthread_join(thread[i], NULL);
            if ( 0 != ret ) {
                fprintf(stderr, "pthread_join: errno=%d¥n", ret);
                exit(1);
            }
        }

        gettimeofday(&et, NULL);
        timersub(&et, &st, &rt);
        printf("%d %d %ld.%ld¥n", malloc_size, malloc_cnt, rt.tv_sec, rt.tv_usec);
        fflush(NULL);
        pthread_barrier_destroy(&barrier);
        for (i=0;i<malloc_cnt*CPU_NUM;i++) {
            free(strp[i]);
        }
        loop++;
    }
    return 0;
}
```

・ チューニング方法

環境変数(XOS_MMM_L_ARENA_LOCK_TYPE)を以下の2つのパターンで設定し、性能を比較します。

1. メモリ獲得性能優先(動的メモリ確保要求を並列処理)

   export XOS_MMM_L_ARENA_LOCK_TYPE=0

2. メモリ使用効率優先(動的メモリ確保要求を逐次処理)

   export XOS_MMM_L_ARENA_LOCK_TYPE=1

各パターンでは、環境変数XOS_MMM_L_ARENA_FREE=2も合わせて設定します。これにより、物理ページ割り当ての処理コストが小さくなり、複数スレッドによる動的メモリ確保要求が競合した場合の処理コストの比較が行いやすくなります。

上記のサンプルプログラムでは、1スレッドあたり、サイズが64KiBのメモリを20480回獲得させています。それを16スレッドで競合させ10回計測します。

- **解説・性能予測**

  複数のスレッドが同時にmalloc(3)を発行すると、malloc(3)によるメモリ獲得処理が競合する可能性が高くなります。

  XOS_MMM_L_ARENA_LOCK_TYPE=0に設定した場合、malloc(3)競合の発生時にスレッドヒープ領域を生成することで動的メモリ確保要求を並列処理でき、メモリ獲得にかかる時間が短くなると考えられます。

  XOS_MMM_L_ARENA_LOCK_TYPE=1に設定した場合、malloc(3)競合の発生時にスレッドヒープ領域を生成せずプロセスヒープ領域を共有します。そのため、現在処理中のmalloc(3)処理の完了を待ち合わせて逐次処理をしなければならず、メモリ獲得にかかる時間が長くなると考えられます。

3.5 アプリケーションプログラムのデバッグ

本節では、アプリケーションプログラムのメモリの割り当てのデバッグやチューニングに役立つオープンソースソフトウェア(OSS)を紹介します。

- Valgrind

  Vargrindはメモリリークやスレッドエラーを検出するための動的解析ツールです。

  Valgrindはmemcheck, cachegrind, callgrind, helgrindなどの機能で構成されています。

  cachegrindおよびcallgrindの機能は、ラージページライブラリとの同時使用ができますが、それ以外の機能は、mallocをフックするため、ラージページライブラリとの同時使用はできません。

注意

helgrindを使ってアプリケーションプログラムをデバッグするジョブを実行すると、helgrindのプロセスがOOM(Out of memory) killerにより、killされることがあります。

helgrindは、仕様としてプロセス起動時に資源制限値のdata seg size(RLIMIT_DATA)およびstack size(RLIMIT_STACK)に対応するエントリを参照し、その分の仮想メモリを確保しようとします。仮想メモリ空間を構成するためのPTE(ページテーブルエントリ)はシステム用メモリに作成しますが、これらの資源制限値がunlimitedに設定されている場合、helgrindは無限の仮想メモリを確保しようと、許容されたシステム用メモリの量を超えてPTEを作成し、OOMが発生します。

この現象を回避するには、ジョブ実行時に、data seg size(RLIMIT_DATA)およびstack size(RLIMIT_STACK)にそれぞれ対応する、pjsubコマンドのオプションproc-dataおよびproc-stackに適切な資源制限値を指定してください。ジョブ実行時の資源の指定方法の詳細は、マニュアル「ジョブ運用ソフトウェア エンドユーザ向けガイド」の"ジョブ投入時のオプション"の資源の指定に関する記述を参照してください。なお、システムの資源制限値などの制限情報はpjaclコマンドで確認できます。マニュアル「ジョブ運用ソフトウェア エンドユーザ向けガイド」の"制限情報の確認"を参照してください。

参考

Valgrindについては、ValgrindのOSS コミュニティーサイトの情報(http://valgrind.org/)や、RHELの開発者ガイドの情報(https://access.redhat.com/documentation/)を参考にしてください。

- malloc統計情報

  malloc統計情報(malloc_stats(3))は、各領域のメモリ割り当て量を出力するglibcの関数です。

  glibcのmalloc統計情報を利用する際、ソースコードを修正(関数呼び出しを追加)してリビルドする必要があります。

参考

glibcについては、glibcのOSS コミュニティーサイトの情報(https://www.gnu.org/software/libc/)を参考にしてください。

付録A　メッセージ

本章では、HPCタグアドレスオーバーライド制御機能のコマンド(fhetboコマンド)およびラージページライブラリが出力するメッセージについて説明します。メッセージは標準エラー出力に表示します。

A.1　HPCタグアドレスオーバーライド制御機能(fhetboコマンド)

A.1.1　メッセージの読み方

HPC拡張機能のHPCタグアドレスオーバーライド制御機能(fhetboコマンド)が出力するメッセージ形式を以下に示します。

```
メッセージ種別 コンポーネント名 - メッセージ本文
```

HPC拡張機能のメッセージの構成内容を以下に示します。

表A.1 メッセージの構成内容

構成内容	意味
メッセージ種別	以下のメッセージ種別を表示します。 [INFO] インフォメーションメッセージ
コンポーネント名	「xos FHE 番号」を表示します。番号はメッセージ固有の識別番号です。
'-'	区切り文字です。
メッセージ本文	発生したイベントの内容を表示します。

A.1.2　メッセージ

インフォメーションメッセージ

[INFO] xos FHE 1113 - jobid *jobid* hpc tag address override function is enabled with core mask *coremask*.

意味

coremaskで示すコアに対して、タグアドレスオーバーライドが有効(enable)になりました。

jobid:ジョブID

coremask:タグアドレスオーバーライドが有効になったコアのマスク値

下位ビットから0から始まるコア番号順に、有効となったビットに1が立ちます。
例えばcoremaskが"0xfffffffffffff000"の場合、コア12～59番のタグアドレスオーバーライドが有効になったことを意味します。

対処

対処は不要です。

[INFO] xos FHE 1114 - jobid *jobid* hpc tag address override function is disabled with core mask *coremask*.

意味

coremaskで示すコアに対して、タグアドレスオーバーライドが無効(disable)になりました。

jobid:ジョブID

coremask:タグアドレスオーバーライドが無効になったコアのマスク値

下位ビットから0から始まるコア番号順に、無効となったビットに1が立ちます。
例えばcoremaskが"0xfffffffffffff000"の場合、コア12～59番のタグアドレスオーバーライドが無効になったことを意味します。

対処

対処は不要です。

A.2 ラージページライブラリ

A.2.1 メッセージの読み方

ラージページライブラリが出力するメッセージ形式を以下に示します。

```
メッセージ種別 コンポーネント名 - メッセージ本文
```

ラージページライブラリのメッセージの構成内容を以下に示します。

表A.2 メッセージの構成内容

構成内容	意味
メッセージ種別	以下のメッセージ種別を表示します。 [WARN] ワーニングメッセージ [INFO] インフォメーションメッセージ
コンポーネント名	「xos LPG 番号」を表示します。番号はメッセージ固有の識別番号です。
'-'	区切り文字です。
メッセージ本文	発生したイベントの内容を表示します。

A.2.2 メッセージ

警告メッセージ

[WARN] xos LPG 2002 - Failed to map HugeTLBfs for data/bss: *a.out*
The e_type of elf header must be ET_EXEC when using libmpg. You can check it on your load module by readelf -h command.

意味

アプリケーションプログラムのバイナリ形式がPIE(Position Independent Executable)であるため、.data/.bss領域をラージページ(HugeTLBfs)化できません。.data/.bss領域には通常ページを使用して実行を継続しました。

a.out:アプリケーションプログラム

対処

.data/.bss領域をラージページ化する場合、アプリケーションプログラムのバイナリ形式がPIEとならないようにコンパイルしてください。(readelf -hコマンドでアプリケーションプログラムのe_typeがET_EXECになるように作成してください。)

[WARN] xos LPG 2003 - Failed to map HugeTLBfs for data/bss: Layout problem with segments *seg_index1* and *seg_index2*:
Segments would overlap.

意味

アプリケーションプログラムの.data/.bss領域のアドレス範囲が他セグメントと衝突するため、.data/.bss領域をラージページ(HugeTLBfs)化できません。.data/.bss領域にはノーマルページを使用して実行を継続しました。

seg_index1, seg_index2:セグメントインデックス値

対処

.data/.bss領域をラージページ化する場合、アプリケーションプログラムのコンパイル時に適切なリンカスクリプト(リンカオプション)を指定し、.data/.bss領域の仮想アドレスをラージページサイズでアラインしてください。

[WARN] xos LPG 2004 - Failed to map HugeTLBfs for thread stack: specified stack address in pthread_attr: addr=*stack_addr* size=*stack_size*

意味

pthread_create(3)の第二引数のattr(属性)にスレッドスタックの仮想アドレスが指定されているため、スレッドスタックをラージページ(HugeTLBfs)化できません。スレッドスタックにはノーマルページを使用して実行を継続しました。

stack_addr: 指定されたスタックアドレス

stack_size: 指定されたスタックサイズ

対処

スレッドスタックをラージページ化する場合、pthread_create(3)の第二引数のattr(属性)にスレッドスタックの仮想アドレスを指定しないでください。

[WARN] xos LPG 2005 - Failed to map HugeTLBfs for process stack: confirmed the existence of multiple threads.

意味

プロセススタックのラージページ化処理中にマルチスレッドを検出したため、プロセススタックをラージページ(HugeTLBfs)化できません。プロセススタックはノーマルページを使用して実行を継続しました。

対処

プロセススタックのラージページ化処理中はシングルスレッドである必要があります。自身のプログラムでスレッドを生成しているかどうか確認してください。またラージページライブラリ以外の他ライブラリでスレッドを生成している可能性も考えられます。

[WARN] xos LPG 2006 - Failed to map HugeTLBfs for process stack: The bottom of the process stack is invading the next vma: bottom=*stack_bottom_addr* next-vma=*next_vma_addr*

意味

プロセススタックの後方(アドレス高位)のメモリ領域と衝突するため、プロセススタックをラージページ(HugeTLBfs)化できません。プロセススタックにはノーマルページを使用して実行を継続しました。

stack_bottom_addr: スタックボトムアドレス

next_vma_addr: 後方メモリ領域の先頭アドレス

対処

プロセススタックをラージページ化する場合、他のメモリ領域とアドレス範囲が衝突しないことを保証する必要があります。自身のプログラムで明示的にmmap(2)を使用してプロセススタックの付近にメモリ領域をマップしているかどうか確認してください。また、ラージページライブラリ以外の動的ライブラリでメモリ領域をマップしている可能性も考えられます。

[WARN] xos LPG 2007 - Failed to map HugeTLBfs for process stack: The top of the process stack is invading the previous vma: top=*stack_top_addr* prev-vma=*prev_vma_addr*

意味

プロセススタックの前方(アドレス低位)のメモリ領域と衝突するため、プロセススタックをラージページ(HugeTLBfs)化できません。プロセススタックにはノーマルページを使用して実行を継続しました。

stack_top_addr: スタック先頭アドレス

prev_vma_addr: 前方メモリ領域ボトムアドレス

対処

プロセススタックをラージページ化する場合、他のメモリ領域とアドレス範囲が衝突しないことを保証する必要があります。自身のプログラムで明示的にmmap(2)を使用してプロセススタックの付近にメモリ領域をマップしているかどうか確認してください。また、ラージページライブラリ以外の動的ライブラリでメモリ領域をマップしている可能性も考えられます。

インフォメーションメッセージ

[INFO] xos LPG 9999 - fallback alloc has worked.

意味

要求されたサイズのラージページ割り当てに失敗しましたが、環境変数 XOS_MMM_L_HUGETLB_FALLBACK が有効化されているため、ノーマルページ割り当てを行いました。

対処

対処不要です。