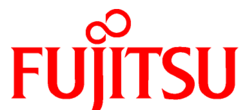


FUJITSU Software
Technical Computing Suite V4.0L20

Job Operation Software
End-user's Guide
for HPC Extensions

J2UL-2453-02ENZ0(04)
September 2022

Preface

Purpose of This Manual

This manual describes the HPC tag address override control function and large page library of the HPC (High Performance Computing) extension function in relation to FX server end-users. This function and library are part of the HPC extension function included in the Job Operation Software of Technical Computing Suite.

Intended Readers

This manual is intended for FX server end-users.

The manual assumes readers have the following knowledge:

- Basic Linux knowledge
- Overall knowledge of the Job Operation Software, obtained from the "Job Operation Software Overview"

Organization of This Manual

This manual is organized as follows.

"Chapter 1 Overview of the HPC Extension Function"

This chapter describes the listed functions provided by the extension function.

"Chapter 2 HPC Tag Address Override Control Function"

This chapter describes the HPC tag address override control function.

"Chapter 3 Large Page Library"

This chapter provides an overview of the memory allocation function and describes the large page function and performance tuning of the FX server.

"Appendix A Messages"

This appendix explains the messages output by the functions in this manual.

Notation in This Manual

Representation of units

The following table lists the prefixes used to represent units in this manual. Basically, disk size is represented as a power of 10, and memory size is represented as a power of 2. Be careful about specifying them when displaying or entering commands.

Prefix	Value	Prefix	Value
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}

Representation of Model Names

In this manual, a computer with a mounted Fujitsu CPU A64FX is abbreviated as "FX server".

Symbols in This Manual

This manual uses the following symbols.

Note

The Note symbol indicates an item requiring special care. Be sure to read these items.

Example

The Example symbol indicates an application example.

See

The See symbol indicates the reference source of detailed information.

Information

The Information symbol indicates reference information.

Export Controls

To export or release this document to a third party, check and take the necessary procedures in accordance with the applicable laws and regulations of your resident country and U.S. export control laws.

Trademarks

Linux(R) is a registered trademark of Linus Torvalds in the U.S. and other countries.

ARM(R) is the registered trademark of ARM Limited in the EU and other countries.

Other company names and product names appearing in this manual are trademarks or registered trademarks of their respective owners.

Date of Publication and Version

Version	Manual Code
September 2022, Version 2.4	J2UL-2453-02ENZ0(04)
November 2021, Version 2.3	J2UL-2453-02ENZ0(03)
August 2021, Version 2.2	J2UL-2453-02ENZ0(02)
September 2020, Version 2.1	J2UL-2453-02ENZ0(01)
March 2020, 2nd version	J2UL-2453-02ENZ0(00)
January 2020, First version	J2UL-2453-01ENZ0(00)

Copyright

Copyright FUJITSU LIMITED 2020-2022

Update history

Changes	Location	Version
Added a note about the default behavior of the large page library.	3.3.1	2.4
Also fixed errata.	-	
Changed the description of the XOS_MMM_L_ARENA_LOCK_TYPE environment variable.	3.4.3	2.3
Changed descriptions of messages for HPC tag address override control function.	A.1	2.2
Added descriptions of the large page library link order.	3.3.1	2.1
Improved the description of conditions under which cache coloring is enabled.	3.4.3	

Changes	Location	Version
Changed the look according to product upgrades.	-	2

All rights reserved.
The information in this manual is subject to change without notice.

Contents

Chapter 1 Overview of the HPC Extension Function....1
- 1.1 HPC Extension Function List....1

Chapter 2 HPC Tag Address Override Control Function....2
- 2.1 HPC Tag Address Override Control Function....2
  - 2.1.1 HPC Tag Address Override Function....2
  - 2.1.2 fhetbo Command....2

Chapter 3 Large Page Library....4
- 3.1 Overview of the Memory Allocation Function....4
  - 3.1.1 Large Page Function....4
    - 3.1.1.1 Memory Address Translation and TLB....4
    - 3.1.1.2 Normal Page and Large Page....5
    - 3.1.1.3 Benefits and Drawbacks by Differences in Page Size....6
  - 3.1.2 Paging Methods....6
- 3.2 FX Server Memory Configuration and Memory Allocation....8
  - 3.2.1 System Memory and Job Memory....8
  - 3.2.2 Job Memory Division Function....8
- 3.3 Large Pages of the FX Server....9
  - 3.3.1 FX Server Memory Areas and Large Page Applicability....9
  - 3.3.2 Memory Areas for Placing Application Program Data....11
- 3.4 Environment Variables for Large Page Library Settings....13
  - 3.4.1 Basic Settings of the Large Page Library....13
  - 3.4.2 Paging Method Setting....15
  - 3.4.3 Settings for Tuning....15
- 3.5 Debugging an Application Program....23

Appendix A Messages....25
- A.1 HPC Tag Address Override Control Function (fhetbo Command)....25
  - A.1.1 How to Read Messages....25
  - A.1.2 Messages....25
- A.2 Large Page Library....26
  - A.2.1 How to Read Messages....26
  - A.2.2 Messages....26

Chapter 1 Overview of the HPC Extension Function

This chapter describes the function (HPC extension function) supporting the use of FX server-specific functions in the OS and Technical Computing Suite.

1.1 HPC Extension Function List

This section lists the end-user functions provided by the HPC extension function.

- HPC tag address override control function
  It provides support for A64 FX processor-specific performance tuning for applications and obtaining hardware event information with the profiler.
- Large page library
  The FX server efficiently uses Huge Pages (HugeTLBfs) with the library.

The HPC extension function operates in linkage with individual functions provided by the Job Operation Software of Technical Computing Suite. For the positioning of the HPC extension function in the Job Operation Software of Technical Computing Suite, see the "Job Operation Software Overview" manual.

Chapter 2 HPC Tag Address Override Control Function

This chapter describes the HPC tag address override control function of the HPC extension function.

2.1 HPC Tag Address Override Control Function

This section describes the HPC tag address override feature specific to the A64FX processor and the user command that controls it.

2.1.1 HPC Tag Address Override Function

The HPC tag address override function is a feature specific to the A64FX processor used in the compute nodes of the FX server to control the performance flags.

This performance flag is controlled by the configuration associated with the ARMv8-A Address tagging feature of the A64FX processor. The ARMv8-A Address tagging feature is an ARMv8 processor feature that prevents the hardware (MMU: Memory Management Unit) from interpreting the upper 8 bits of the virtual address.

On FX servers, the unique HPC tag address override function is enabled, while the ARMv8-A Address tagging function is disabled.

For this reason, if an FX server user creates an application program with the intention of using the ARMv8-A Address tagging function and executes a job, an unintended performance-related flag is set and an unintended behavior such as performance degradation may occur.

Thus, for users who want to use the ARMv8-A Address tagging feature, we provide a user command (HPC tag address override control function) to control the HPC tag address override feature.

2.1.2 fhetbo Command

The fhetbo command enables or disables the HPC tag address override function as the HPC tag address override control function.

If you disable the HPC tag address override feature, you can use the ARMv8-A Address tagging feature.

NAME

fhetbo - Controls the HPC tag address override function.

SYNOPSIS

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

OPTIONS

enable: Enable the HPC tag address override function.

disable: Disable the HPC tag address override function.

DESCRIPTION

This command enables or disables the HPC tag address override function for all the cores allocated to a job.

The specification of an option is required. If no option is specified, the command displays how to use the command.

The command can be used only within a job.

In the assumed usage scenario, the command is called in a job script.

The command is executed in units of nodes, and the affected range of the command is limited to the cores allocated to the same job within the node executing the command. Therefore, when used with a multinode job, the command must be executed via the mpiexec command.

There is no special restriction on the number of command calls or state overwriting within a job. The instruction in the last executed command is valid, and you can repeatedly change the enable/disable option. After the job ends, the HPC tag address override function returns to the enabled state.

The result of enabling/disabling the HPC tag address override function is output as a message.

For details, see "Appendix A Messages."

An example of a created job script is shown below.

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

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

mpiexec -n $NR_PROCS --nompi /opt/FJSVxos/fhehpc/bin/fhetbo disable   <- Disable HPC tag address
override function
mpiexec -n $NR_PROCS ./a.out                                          <- Execute program a.out
mpiexec -n $NR_PROCS --nompi /opt/FJSVxos/fhehpc/bin/fhetbo enable    <- Enable HPC tag address
override function
```

For details on the job execution environment and how to code a job script for the Job Operation Software of Technical Computing Suite, see the "Job Operation Software End-user's Guide" manual.

For details on the mpiexec command, see the "MPI User's Guide", a Development Studio manual.

Chapter 3 Large Page Library

This chapter describes the large page library that is part of the HPC extension function.

Some application programs use a large amount of memory in the HPC area. The cost of memory management by the OS affects the execution performance of these application programs. To reduce this cost, the HPC extension function provides an original extended function for large pages.

The chapter also describes the basic memory allocation ideas needed to understand the large page library, the memory configurations for the FX server (job memory division function), and more.

3.1 Overview of the Memory Allocation Function

This section describes the following function and processing as an overview of the memory allocation function, which is an extension function of the FX server:

- Large page function

- Paging methods

3.1.1 Large Page Function

The large page function extends HugeTLBfs, a standard function in Linux, and allocates memory (large page) with a larger page size than a normal page to application programs that handle large amounts of data. By doing so, it reduces the cost of address translation processing by the OS and increases memory access performance.

3.1.1.1 Memory Address Translation and TLB

Application programs use virtual memory addresses to access memory. Therefore, virtual memory addresses must be translated into physical memory addresses. An address translation table in main memory is used to translate the addresses. At this time, to further increase the access speed, the load-and-store mechanism in the CPU uses an address translation buffer called the TLB (Translation Look-aside Buffer) to search for a physical address.

The load-and-store mechanism in the CPU receives load-and-store requests for memory from running application programs. "Figure 3.1 Translation From a Virtual Memory Address to a Physical Memory Address" shows an outline of memory address translation using the load-and-store mechanism.

1. The mechanism searches for the physical memory address corresponding to the virtual memory address specified as the load-and-store destination from the TLB.

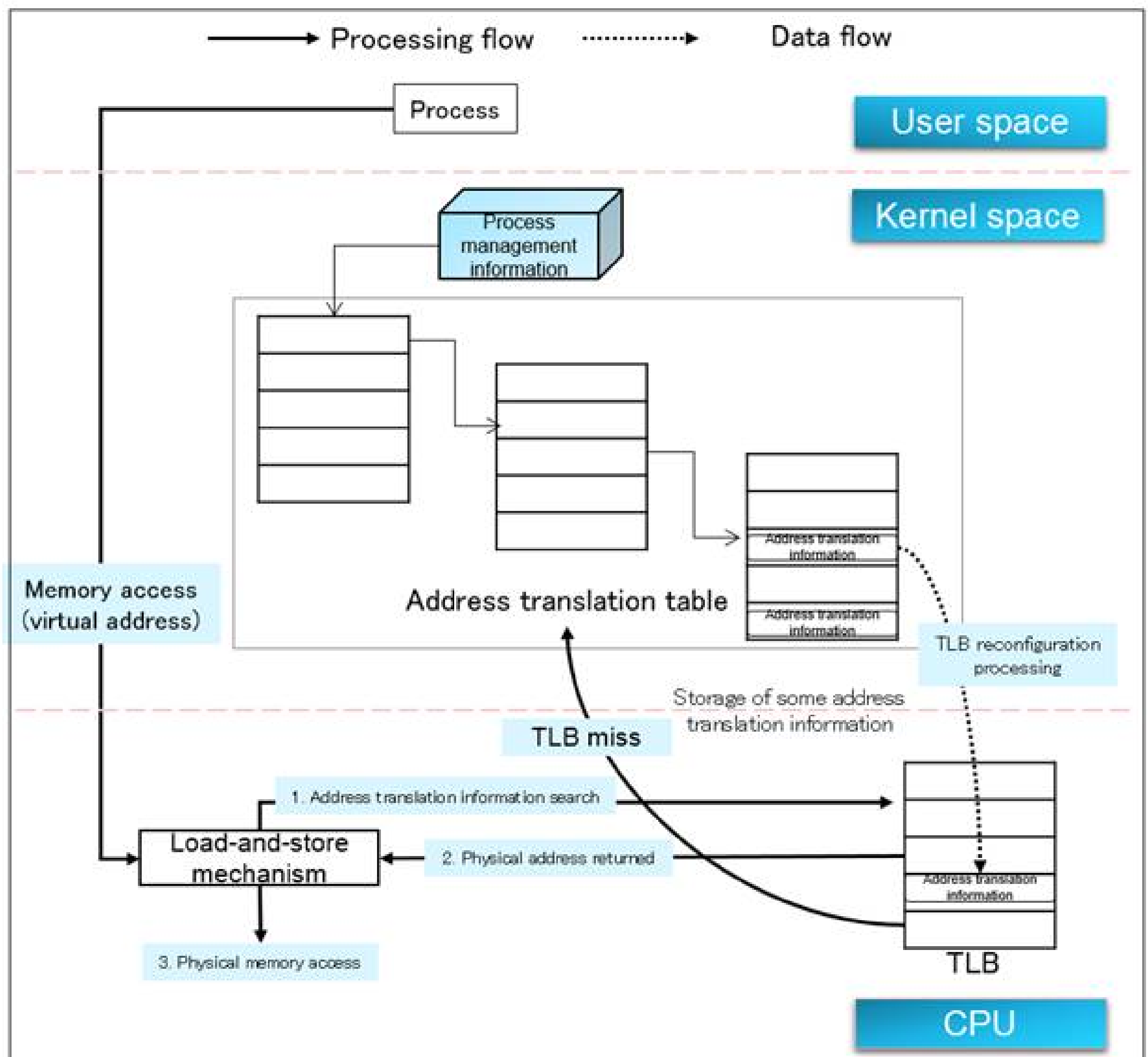
   If the TLB does not have corresponding address translation information, a TLB miss occurs and the TLB searches for and reads in the address translation information from the address translation table.

2. The mechanism obtains the physical memory address from the TLB using the corresponding address translation information.

3. Physical memory access to the load-and-store destination begins.

A TLB miss occurs when the physical memory address is not found in the TLB. If a TLB miss occurs, TLB reconfiguration processing (reading the address translation table again) is necessary. Generally, the cost (time) required for this processing is large.

Figure 3.1 Translation From a Virtual Memory Address to a Physical Memory Address



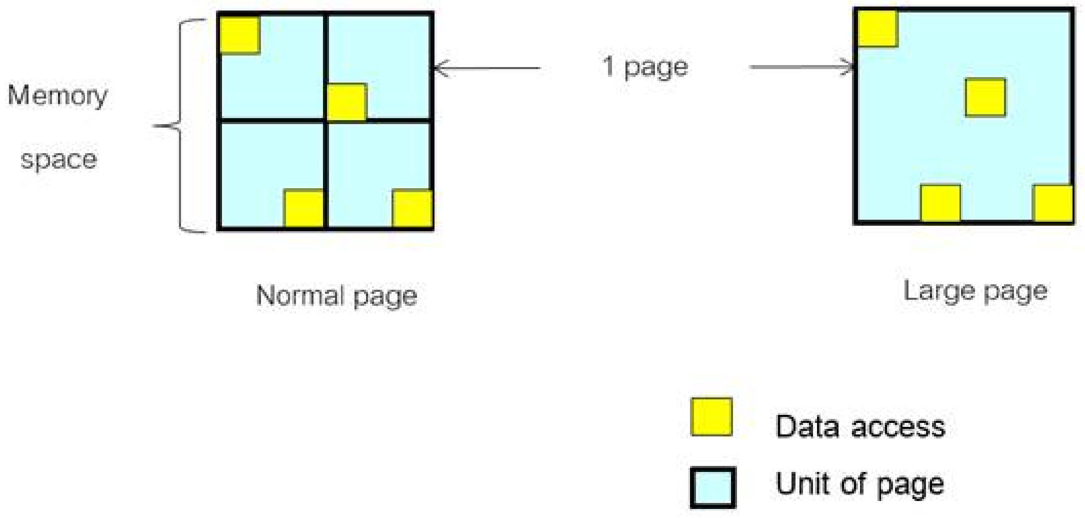
3.1.1.2 Normal Page and Large Page

The OS of the FX server can use page sizes of 64 KiB for a normal page and 2 MiB for a large page. The large page function has large pages enabled by default. You can select to enable or disable large pages by using an environment variable (XOS_MMM_L_HPAGE_TYPE) described in the section titled "3.4 Environment Variables for Large Page Library Settings."

If an application program handles a large amount of memory with normal pages, it accesses many pages, and TLB reconfiguration processing tends to occur frequently. If the application program uses large pages, it accesses fewer pages as compared to normal pages, and TLB reconfiguration processing occurs less frequently.

This means you can reduce TLB misses by allocating large pages to application programs that use a large amount of memory. In turn, this reduces the cost of TLB reconfiguration processing by the OS and improves memory access performance.

Figure 3.2 Conceptual Illustration of Data Access on Normal and Large Pages



A TLB miss occurs at the first access to a memory address by an application program, because the TLB does not have translation information for the address. Looking at "Figure 3.2 Conceptual Illustration of Data Access on Normal and Large Pages" as an example, the data access to four normal pages in a certain memory space caused four TLB misses, whereas the same data access to a large page in the same memory space caused only one TLB miss.

3.1.1.3 Benefits and Drawbacks by Differences in Page Size

The following table shows the respective benefits and drawbacks of using page sizes of 64 KiB and 2 MiB.

Table 3.1 Benefits and Drawbacks According to Page Size

Evaluation Item	64 KiB	2 MiB
TLB miss rate	High	Low
Memory initialization cost	Low	High
Memory use efficiency	High	Low

1. Normal pages have a page size of 64 KiB. Although their TLB miss rate is high, the memory initialization cost is low and memory use efficiency is high, so they may be effective when memory usage by application programs is small.

2. Large pages have a page size of 2 MiB. Their memory initialization cost is higher and memory use efficiency is lower as compared to 64-KiB pages. However, they have a lower TLB miss rate, so the large page function of the FX server uses large pages (2 MiB) by default.

Application programs use memory of various sizes ranging from small-scale memory for, as an example, a communication buffer used in typical communications to large-scale memory for computing.

Memory use efficiency may be low for large pages used when memory usage by application programs is small. For example, if large pages are used even when the memory usage of the heap area is only 1 MiB, 2 MiB of memory is reserved, resulting in an unused memory area of 1 MiB.

On the other hand, the TLB miss rate is high for normal pages used when memory usage by application programs is significant. Furthermore, there will be a high possibility that execution performance does not improve.

3.1.2 Paging Methods

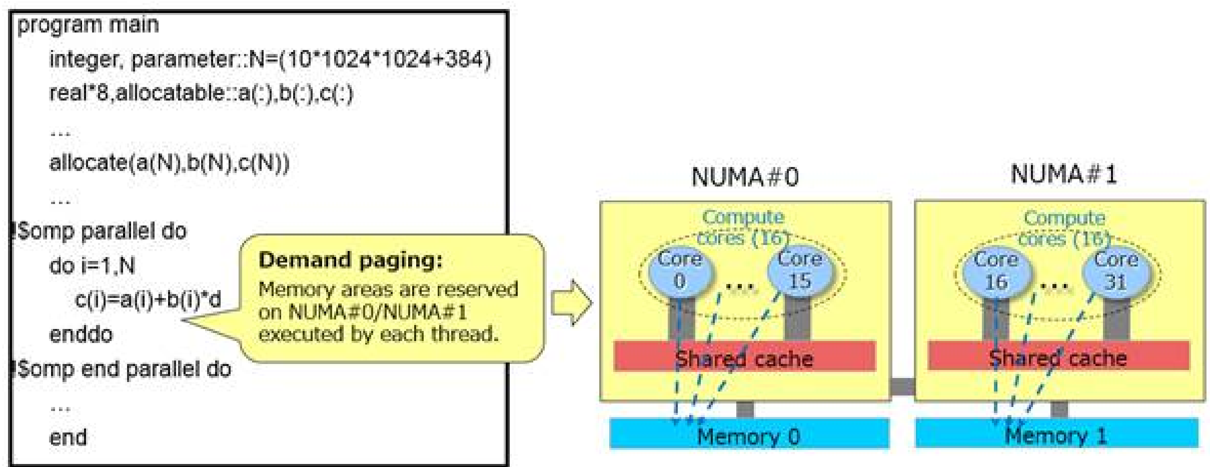
The two paging methods are demand paging and prepaging. For the large page function of the FX server, you can set a paging method for each of the following memory areas: static data area (.bss area), stack/thread stack areas, and reserved dynamic memory areas. This uses an environment variable (XOS_MMM_L_PAGING_POLICY) described in the section titled "3.4 Environment Variables for Large Page Library Settings."

1. The demand paging method is a method of allocating a page to main memory, as needed, if the necessary page is not in main memory during execution of an application program. The physical page is allocated at the timing of initial access to the memory area.
2. The prepaging method is a method of allocating a page to main memory beforehand. The physical page is allocated at the timing of memory area allocation.

The examples of simple NUMA configurations in "Figure 3.3 Memory Allocation Differences From Paging Method Differences" show differences in the operation of an application program because of paging method differences. The following example assumes 1 CPU (32 cores) has 2 NUMAs named NUMA#0 and NUMA#1.

For details on the actual NUMA configuration of the FX server, see "3.2.2 Job Memory Division Function."

Figure 3.3 Memory Allocation Differences From Paging Method Differences



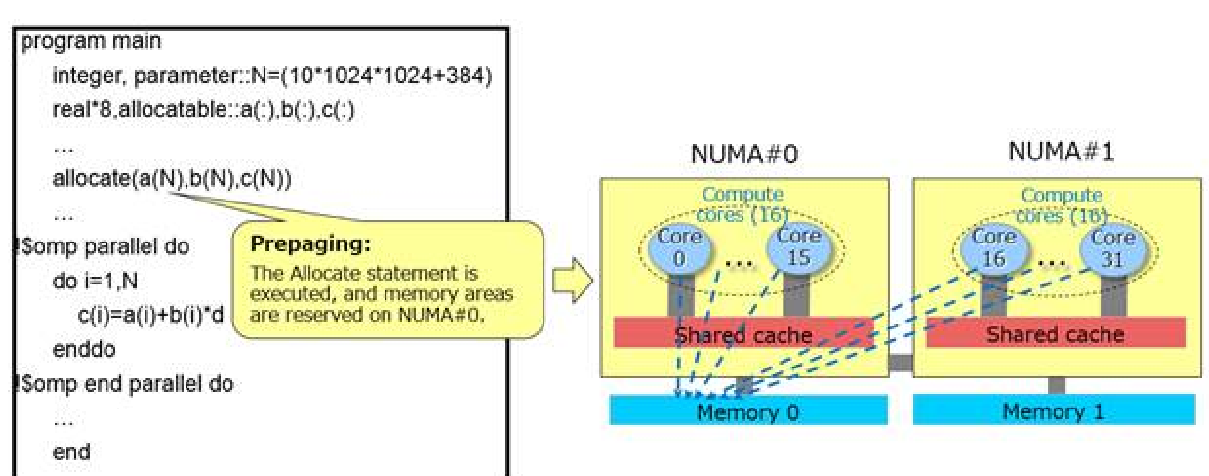


Table 3.2 Memory Access Costs According to Paging Method Differences

Paging Method	Memory Access Inside/Outside NUMA	Initial Memory Access
Demand paging method	Uses internal NUMA memory as much as possible (Cost: Low)	Loads page to physical memory during memory access (Cost: High)
Prepaging method	Uses memory regardless of whether it is inside or outside NUMA (Cost: High)	Loads page to physical memory beforehand (Cost: Low)

As shown in "Table 3.2 Memory Access Costs According to Paging Method Differences," two factors affect memory access performance. The first factor is whether the physical memory accessed from a certain compute core is in the same NUMA. Access to physical memory

not belonging to the same NUMA as the compute core requires a greater processing cost than access to physical memory in the same NUMA. The second factor is the cost when memory is accessed for the first time. If the demand paging method is selected, physical memory is allocated at the memory access time, so a page load processing cost is incurred when the memory is accessed for the first time.

For example, suppose that a four-core thread-parallel application program dynamically acquires memory areas for individual threads. In this case, switching from the prepaging method to the demand paging method increases the frequency of access to physical memory in the same NUMA as the compute core. You can expect an improvement in memory access performance as a result.

3.2 FX Server Memory Configuration and Memory Allocation

In order for the FX server to secure as many large pages as necessary for jobs, NUMA nodes are virtually divided into system nodes and job nodes by a job memory division function. This section describes the job memory division function.

3.2.1 System Memory and Job Memory

In each compute node, memory usage is calculated separately between memory used by application programs (job memory) and memory used by the system (system memory). Large pages are used only with job memory.

- System memory

  The memory is used for IO processing pages (page cache) and operations of the OS itself. The size of an allocated page is 64 KiB, and the allocation method is the demand paging method.

- Job memory

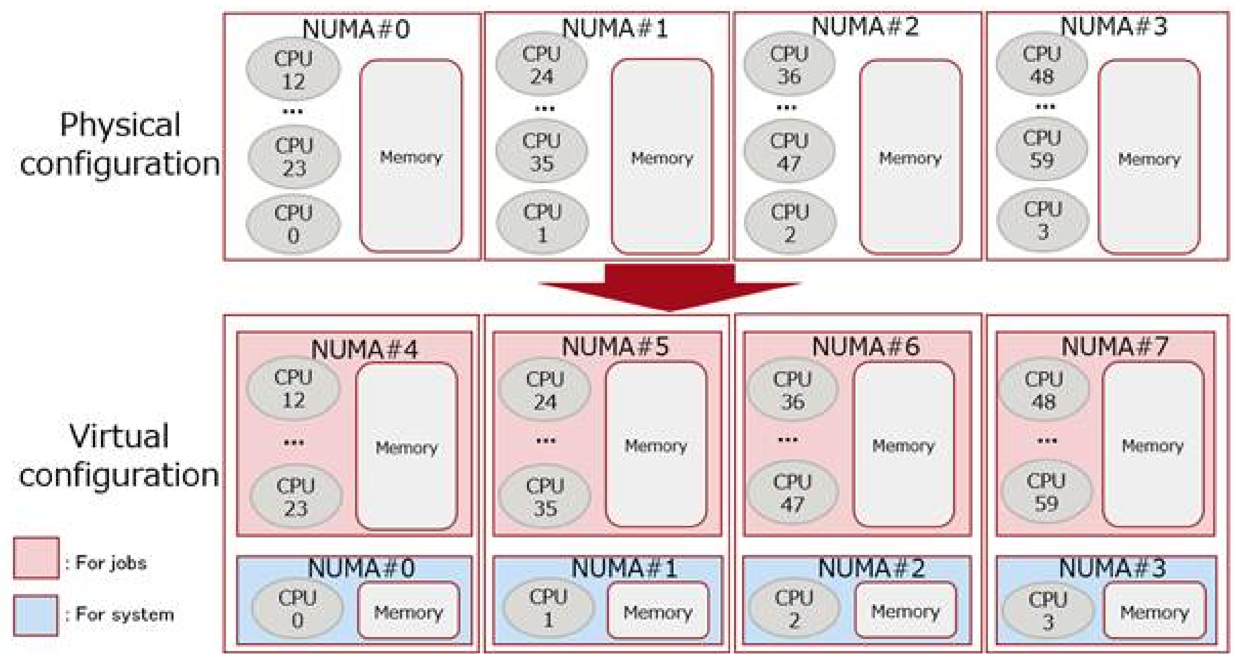
  The memory is allocated for data used by application programs. The FX server uses large pages with a size of 2 MiB. End-users can specify a paging method by using an environment variable described in the section titled "3.4 Environment Variables for Large Page Library Settings."

3.2.2 Job Memory Division Function

The FX server has a configuration with four physical NUMA nodes, and each physical NUMA node is virtually further divided into two NUMA nodes: system node and job node. (NUMA#0 to #3 are allocated to system memory, and NUMA#4 to #7 are allocated to job memory.)

In this way, memory access in the processing of jobs executed by end-users is kept separate, so it is not affected by memory access in system processing. This is expected to improve job execution performance.

Figure 3.4 NUMA Node Configuration of the FX Server



The application programs executed by end-users as jobs use CPU12 to CPU59 in NUMA#4 to NUMA#7. Job memory is allocated to these CPUs. (The system uses CPU0 to CPU3. CPU4 to CPU11 are unassigned.)

In a multithread application program, the processing that assigns processes (threads) using the CPU numbers obtained by sched_getaffinity(2) operates normally without any change. However, if the processing uses static CPU numbers without consideration of the configuration, the fact that CPU numbering starts at 12 must be taken into account in the coding of the program.

For an example of program coding with consideration of the FX server configuration in CPU numbering, see the sample program ("Example of checking the effect of tuning with the environment variable XOS_MMM_L_ARENA_LOCK_TYPE") in "3.4.3 Settings for Tuning."

3.3 Large Pages of the FX Server

The large page library (libmpg.so) is provided as an extension function of the FX server. This section describes the page sizes of individual areas and the location of application program data when using the large page library of the FX server.

3.3.1 FX Server Memory Areas and Large Page Applicability

From the areas listed below, the large page library of the FX server puts the following areas on large pages (2-MiB pages): static data areas (.bss and .data areas), reserved dynamic memory areas (mmap areas), thread heap areas, stack areas, and thread stack areas. Text areas, reserved dynamic memory areas (heap areas), and shared memory areas are not intended to be put on large pages.

Table 3.3 FX Server Memory Areas and Large Page Applicability

Area	Large Page Applicability	Usage of Area
Text (.text) area	No (64-KiB page)	Memory area for placing a series of instructions from an application program (a.out)
Static data (.data) area	Yes (2-MiB page)	Memory area for storing the static data of an application program (a.out) (Initialized)

Area	Large Page Applicability	Usage of Area
Static data (.bss) area	Yes (2-MiB page)	Memory area for storing the static data of an application program (a.out) (Not initialized)
Reserved dynamic memory area (heap area)	No (64-KiB page)	Process heap area/Main thread heap area This memory area is not put on large pages in the large page library. Memory areas are allocated when:. Acquired with a brk(2)/sbrk(2) system call, or allocated when a dynamic memory reservation request (malloc(3)) specifying a smaller size than the size specified in the environment variable MALLOC_MMAP_THRESHOLD_ is issued without linking the large page library.
Thread heap area	Yes (2-MiB page)	Subthread heap area
Stack area	Yes (2-MiB page)	Process stack area/Main thread stack area (Use of large pages must be explicitly specified in the environment variable XOS_MMM_L_LPG_MODE.)
Thread stack area	Yes (2-MiB page)	Subthread stack area (Use of large pages must be explicitly specified in the environment variable XOS_MMM_L_LPG_MODE.)
Reserved dynamic memory area (mmap area)	Yes (2-MiB page)	Memory area allocated when mmap(2) is issued Memory areas are allocated when:. Allocated when the large page library is linked and a dynamic memory reservation request (malloc(3)) specifying a larger size than the size specified in the environment variable MALLOC_MMAP_THRESHOLD_ is issued without linking the large page library, or alternatively allocated as a thread heap area/ thread stack area too.
Shared memory	No (64-KiB page)	Used for memory sharing between processes

The large page library, libmpg.so, is a large page function optimized for the FX server. By using the overcommit function of HugeTLBfs, it improves the usability of the user I/F without reserving a memory pool beforehand. By specifying the -Klargepage option for the Development Studio compiler, the application program will use this library. To use the standard large page function in Linux without using libmpg.so, specify the -Knolargepage option for the Development Studio compiler. To use normal pages, explicitly specify the environment variable XOS_MMM_L_HPAGE_TYPE=none.

For details on the environment variables described in this section, see the section titled "3.4 Environment Variables for Large Page Library Settings."

See

For details on how to compile a program that uses normal or large pages in Development Studio (-K{largepage|nolargepage}), see the following Development Studio manuals: "Fortran User's Guide", "C User's Guide", and "C++ User's Guide".

- Link order of the large page library

  The large page library specification (-lmpg option specified by the Development Studio compiler) must always precede the library explicitly specified by the end user (e.g., -lc, -lpthread, etc.). Otherwise, large pages may not be allocated successfully.

- Memory area acquired by brk(2)/sbrk(2)

  Due to kernel specifications, memory acquired by brk(2)/sbrk(2) is not on large page. So linking the large page library libmpg.so together with brk(2)/sbrk(2) and malloc(3)(Size request greater than environment variable MALLOC_MMAP_THRESHOLD_)

would result in a mixture of normal and large pages, and depending on the amount and timing of memory acquisition, brk(2)/sbrk(2) could end with ENOMEM without getting any memory.

It is recommended that you use malloc(3) to link the large page library libmpg.so.

If you want to use brk(2)/sbrk(2) in application programs that link the large page library libmpg.so, you must explicitly specify the environment variable XOS_MMM_L_HPAGE_TYPE=none.

- Applications that signal frequently

  If you link a large page library(libmpg) with a prepaging method and use the timer_create(2) system call to generate a SIGALRM/SIGVTALRM or any other signal frequently, the application-issued fork(2)(clone(2)) receives the signal and the system call continues to return with ERESTARTNOINTR. As a result, frequent retries of fork(2)(clone(2)) may occur and it does not terminate.

  In such cases, setting the paging method to demand paging avoids the problem.

  Run the application with the environment variable XOS_MMM_L_PAGING_POLICY=demand:demand:demand explicitly.

- Default behavior of the large page library

  The default behavior of the large page library is to temporarily use twice the combined byte size (*1) of "Static data (.data) area" and "Static data (.bss) area" when starting the application program (a.out).
  Therefore, if an application program has a large static data area, it may be forced to terminate with SIGKILL during startup due to insufficient memory.
  (*1) These sizes can be checked with the readelf -S command. It temporarily uses twice this size of memory.

  In this case, you might be able to avoid it by changing the default behavior of the large page library. You can reduce the amount of memory used when a program starts by using demand paging as the default paging method for "Static data (.bss) area".

  The following example sets the environment variable XOS_MMM_L_PAGING_POLICY with the -x option of the pjsub command to change the paging method for the "Static data (.bss) area".

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

  For details on the XOS_MMM_L_PAGING_POLICY environment variable, see "3.4.2 Paging Method Setting".

You can create large-page-enabled application programs, even with a general compiler (e.g., gcc), by linking this large page library, libmpg.so. If you explicitly link to other libraries (example below, "-lc -lpthread"), specify that the large page libraries always precede those libraries.

To enable large pages, the large page library also provides a large page linker script in the path shown below. This script is intended to put static data (.data and .bss) on large pages. Specify the linker script appropriately for the compiler too.

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

The following example compiles an application program on gcc.

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

If an application program is compiled in PIE (Position Independent Executable) format, the .data/.bss areas are not on large pages. In this case, the large page library only outputs a warning log, normal pages are used for the .data/.bss areas, and the application program continues running. As an example, to compile an application program on gcc while explicitly avoiding the PIE format, use the -no-pie option.

3.3.2 Memory Areas for Placing Application Program Data

Variables in application programs are placed in different memory areas according to how they are defined. Therefore, when specifying the page size of each memory area, you need to be aware of which memory areas that the variables are placed in.

This section describes which memory areas have variables placed in them along with how they are declared in examples of Fortran, C, and C++ source code. The page size for each allocated area is as shown in "Table 3.3 FX Server Memory Areas and Large Page Applicability " in "3.3.1 FX Server Memory Areas and Large Page Applicability."

- Fortran

```
program main
integer*8, parameter::N=(1024_8)
real*8 a(N)                        ! a: Local array without initial values
real*8 :: b(N)=1.0                 ! b: Array with initial values
real*8,allocatable::c(:)           ! c: Allocatable array
allocate(c(N))
...
end
```

Local array a is placed in a static data area (.bss). However, for an application program compiled with the -Kauto or -Kthreadsafe enabling option, it is placed in the process stack area. Array b is placed in a static data area (.data). Array c is placed in a reserved dynamic memory area (heap or mmap area).

- C

```
#define N 1024
double a[N];                          // a: Global variable without initial value
double b[N]={1.0};                    // b: Global variable with initial value
double *c;
double *d;                            // c, d: Pointer variable
void *func_thread(void *args) {
  double f[N];                        // f: Local variable
  d=(double *)malloc(sizeof(double)*N);
  ...
}
int main(void){
  double e[N];                        // e: Local variable
  c=(double *)malloc(sizeof(double)*N);
  …

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

Global variable a is placed in a static data area (.bss). Global variable b is placed in a static data area (.data). Pointer variables c and d are placed in a reserved dynamic memory area (heap or mmap area). Local variable e is placed in the process stack area. Local variable f is placed in the thread stack area.

Note

When the thread-parallel application program in each programming language is executed, a dedicated thread stack area is prepared as the stack area for each thread.

- C++

```
const int N = 1024;
struct Klass {
  double k;
  Klass() : k (0.0) {}
  Klass(double K) : k (K) {}
};
std::vector<Klass> a(N);          // Secure area by using vector class
```

```
int main(){
    Klass* b = new Klass[N];        // Secure area by using new operator
    std::vector<double> c(N);       // Secure area by using vector class
    return 0;
}
```

Variables a, b, and c are all placed in a reserved dynamic memory area (heap or mmap area).

The memory areas allocated for the variables in C++ are the same as those in C, but the memory areas dynamically acquired by vector class, etc. are allocated as reserved dynamic memory areas (heap area or mmap area).

3.4 Environment Variables for Large Page Library Settings

You can use the environment variables shown in this section to adjust the behavior of the large page library.

3.4.1 Basic Settings of the Large Page Library

This section shows the environment variables for the basic settings of the large page library.

Table 3.4 Environment Variables for the Basic Settings of the Large Page Library

Variable Name	Setting Value	Default Value	Details
XOS_MMM_L_HPAGE_TYPE	hugetlbfs|none	hugetlbfs	The setting is the selection to enable/disable operations with large page allocation by the large page library. "hugetlbfs" enables large pages with HugeTLBfs. "none" disables large pages with the large page library. If it is specified, all specified environment variables beginning with "XOS_MMM_L_" for the large page library are invalid. If any other value is specified, "hugetlbfs" is assumed specified. There is a precaution on enabling large pages. See the note below this table ("Stack area display of /proc/*pid*/maps when large pages are enabled").
XOS_MMM_L_LPG_MODE	base+stack|base	base+stack	The setting is the selection to enable/disable operation with large page allocation for the stack area and thread stack area. "base+stack" enables large pages for not only the static data and reserved dynamic memory areas but also the stack area and thread stack area. "base" enables large pages for only the static data and reserved dynamic memory areas. Large pages are not enabled for the stack area and thread stack area. If any other value is specified, "base+stack" is assumed specified.

Variable Name	Setting Value	Default Value	Details
			There is a precaution on enabling large pages for the stack area. See the note below this table ("Alignment with the stack area on large pages").
XOS_MMM_L_PRINT_ENV	on|off|1|0	0	The setting is used for debugging application programs. If "1" or "on" is specified in this environment variable, a list of the performance tuning environment variables provided by the large page library is output to the standard error output. The list is output only once, at the application program start time (before the main function is executed). If "0" or "off" is specified, the list of performance tuning environment variables is not output to the standard error output. If any other value is specified, "0" is assumed specified. The output after the execution of the main function does not reflect the settings changed by mallopt(3).

Note

- Stack area display of /proc/*pid*/maps when large pages are enabled (*pid*: Process ID)

  The displayed contents of /proc/*pid*/maps are partly rewritten when large pages are enabled. This is shown in the following examples. (The same applies to /proc/*pid*/numa_maps.)

  - Static data (.bss/.data area) or reserved dynamic memory area (heap/mmap area)

    (Before enabling large pages)

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

    (After enabling large pages)

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

  - Stack area (process stack area/main thread stack area)

    (Before enabling large pages)

    ```
    ffffffd0000-1000000000000 rw-p 00000000 00:00 0             [stack]
    ```

    (After enabling large pages)

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

- Alignment with the stack area on large pages

  Basically, the size of a stack area on a large page is aligned to the size of an allocated HugeTLBfs page, based on the soft limit value that is set in ulimit -s.

  The start and end addresses of an allocated HugeTLBfs page are also similarly aligned up or down.

  As a result of aligning mapped addresses and sizes, the stack area may overlap an adjacent area. If there is an overlap, large pages are disabled. Then, a warning message is output, and application programs continue as is with normal pages.

3.4.2 Paging Method Setting

This section shows the environment variable for a paging method setting of the large page library.

Table 3.5 Environment Variable for a Paging Method Setting

Variable Name	Setting Value	Default Value	Details
XOS_MMM_L_PAGING_POLICY	[demand|prepage]: [demand|prepage]: [demand|prepage]	prepage:demand:prepage	The setting is the selection of a paging method (page allocation time) for each memory area. "demand" means the demand paging method, and "prepage" means the prepaging method. This variable specifies paging methods for three memory areas by delimiting the methods with a colon (:). The first specified method is for the .bss area for static data. ("prepage" is always set for the .data area for static data, which is not subject to specification of a paging method.) The second specified method is for the stack area and thread stack area. The third specified method is for the reserved dynamic memory areas. If any other value is specified, "prepage:demand:prepage" is assumed specified. The prepaging method prevents subsequent occurrences of page faults in an application program and tends to improve performance and also reduce performance variations. However, in some cases, it is better not to select the prepaging method for any application program that implements prepaging with its own mechanism (for example, zero-clearing or the like) or accesses only a partial memory area.

3.4.3 Settings for Tuning

This section shows the environment variables for tuning related to large page allocation.

Of these environment variables, those with names beginning with "XOS_MMM_L_" have uniquely been added to the large page library. The other environment variables are glibc variables.

Table 3.6 Environment Variables for Tuning

Variable Name	Setting Value	Default Value	Details
XOS_MMM_L_ARENA_FREE	1|2	1	The setting relates to handling of the heap area released by free(3). If "1" is specified, the memory that can be released is immediately released. If "2" is specified, no memory is released, and all the memory is reused in pools. If a value other than "1" or "2" is specified, "1" is assumed specified. If "2" is specified, memory is allocated from the heap area even for a memory request for a size equal to or greater than the value specified in

Variable Name	Setting Value	Default Value	Details
			MALLOC_MMAP_THRESHOLD_. Even after being released, the memory is retained as a free memory area. Release processing of the heap area is not executed. Since memory of all sizes must be allocated in a single heap area to prevent decreases in memory use efficiency, the thread heap area is not generated. In other words, "2" is equivalent to the combination of the following settings: 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	The setting relates to the memory allocation policy. "0" means that memory acquisition performance takes priority, whereas "1" means memory use efficiency takes priority. If a value other than "0" or "1" is specified, "1" is assumed specified. If "0", a new thread heap area is generated when the main arena contention occurs. If "1", when there is a main arena contention, it will generate a thread heap area for each thread if it is within the maximum arena count (XOS_MMM_L_MAX_ARENA_NUM), otherwise it will wait for a lock to be acquired. Suppose that, in a multithread application program, threads simultaneously call a memory acquisition request. In this case, "0" will enable parallel processing but decrease memory use efficiency. "1" will increase memory use efficiency even though memory acquisition will be processed sequentially. In the case of frequent contention among memory acquisition requests from threads in a multithread application program, the "0" setting may improve performance.
XOS_MMM_L_MAX_ARENA_NUM	Integer between 1 and INT_MAX [Decimal number]	1	You can set the number of arenas that can be generated (total number of process heap and thread heap areas) with this variable, which is valid only when XOS_MMM_L_ARENA_LOCK_TYPE=1. Use it when you want to limit the number of thread heap areas generated. With the default setting ("1"), only process heap areas are used, and no thread heap areas are generated. This case is equivalent to XOS_MMM_L_ARENA_LOCK_TYPE=1. If the setting is n (2), in addition to process heap areas, (n-1) thread heap areas may be generated.

Variable Name	Setting Value	Default Value	Details
XOS_MMM_L_HEAP_SIZE_MB	Integer between MALLOC_MMAP_THRESHOLD_ x 2 and ULONG_MAX <in MiB> [Decimal number]	MALLOC_MMAP_THRESHOLD_ x 2	The variable, for using a thread heap area, sets the size of memory acquired when generating or extending the thread heap area. The default value is MALLOC_MMAP_THRESHOLD_ multiplied by 2. A thread heap area with the default value is generated when memory for the thread is acquired for the first time. Furthermore, a thread heap area with the specified value is generated (expanded) when the total amount of acquired memory per thread exceeds the default value. If the expected amount of memory acquired for each thread is small, setting a small value for this environment variable may increase memory use efficiency.
XOS_MMM_L_COLORING	0|1	1	The setting enables/disables cache coloring. Cache coloring reduces conflicts in the L1 cache of the processor. If "0" is specified, cache coloring is not used. If "1" is specified, cache coloring is used when memory is acquired by mmap(2) using the MALLOC_MMAP_THRESHOLD_ size (default value: 128 MiB) or greater. Cache coloring is not used when memory is acquired from the heap area using a smaller size than MALLOC_MMAP_THRESHOLD_. The conditions under which cache coloring must be enabled are as follows: 1) MALLOC_MMAP_THRESHOLD_ or larger size of malloc(3) request, and 2) XOS_MMM_L_FORCE_MMAP_THRESHOLD=1 (This forces mmap(2) to allocate memory) If a value other than "0" or "1" is specified, "1" is assumed specified. Generally, cache coloring tends to improve performance. However, disabling cache coloring with this library can be said to be better for any application program that implements cache coloring with its own mechanism.
XOS_MMM_L_FORCE_MMAP_THRESHOLD	0|1	0	The setting specifies whether mmap(2) takes priority when acquiring memory of a size equal to or greater than MALLOC_MMAP_THRESHOLD_ (default value: 128 MiB). If "0" is specified, mmap(2) does not take priority. First, there is a search for free space in the heap area. Free memory in the heap area is returned when free space is found. mmap(2) is used to acquire memory only when no free space is found in the heap area.

Variable Name	Setting Value	Default Value	Details
			If "1" is specified, mmap(2) takes priority. mmap(2) is used to acquire memory (even if there is free space) without searching for free space in the heap area. If a value other than "0" or "1" is specified, "0" is assumed specified.
MALLOC_CHECK_	0|1|2|3|5|7 [Decimal number]	3	The setting relates to detection of programming errors (memory destruction, double free, etc.). According to the setting, the following actions are taken when errors are detected. 0: Ignore the error, and continue processing. 1: Output a detailed error message, and continue processing. 2: Abort the application program. 3: Output a detailed error message, stack trace data, and memory mapping data, and abort the application program. 5: Output a simple error message, and continue processing. 7: Output a simple error message, stack trace data, and memory mapping data, and abort the application program. If any other value is specified, action is taken according to the lower 3 bits of the value as a binary number. Not all errors can be detected. Errors such as a memory leak cannot be detected.
MALLOC_TOP_PAD_	Integer between 0 and ULONG_MAX <in bytes> [Decimal number]	131072 (= 128 KiB)	The setting specifies the extended size for each time that the heap area is extended. A value rounded up to the nearest page size is used. If this setting value exceeds the amount of memory acquired or released at one time by an application program, the number of issued system calls decreases, and memory acquisition performance may improve. (However, memory use efficiency may decrease.)
MALLOC_PERTURB_	Integer between INT_MIN and INT_MAX [Decimal number]	0	The setting is used for debugging application programs. Based on the value specified in this environment variable, a memory area is filled when memory is acquired (except by calloc(3)) or released. When memory is acquired, the complement of the lowest byte of the specified value is written. When memory is released, the lowest byte of the specified value is written. Use this variable to detect problems like the use of an uninitialized memory area or a reference to a released memory area.
MALLOC_MMAP_MAX_	Integer between INT_MIN and INT_MAX	2097152 (= 2*1024*1024)	The setting is the value of the upper limit on the number of times that memory is acquired by mmap(2) during acquisition of memory of a size

Variable Name	Setting Value	Default Value	Details
	[Decimal number]		equal to or greater than MALLOC_MMAP_THRESHOLD_ (default size: 128 MiB). The current count value is incremented by 1 when mmap(2) acquires memory. The current count value is decremented by 1 when free(3) releases memory. An upper limit value is set for each process. Set "0" to not acquire memory with mmap(2) but instead acquire memory from the heap area regardless of the memory acquisition size and set value of MALLOC_MMAP_THRESHOLD_.
MALLOC_MMAP_THRESHOLD_	Integer between 0 and ULONG_MAX <in bytes> [Decimal or hexadecimal number]	134217728 (= 128 MiB)	For a memory acquisition request with a memory size equal to or greater than the size specified in this environment variable, mmap(2) is used to acquire memory. For a memory acquisition request with a memory size smaller than the size specified in the environment variable, memory is acquired from the heap area. When executed for memory acquired with mmap(2), free(3) immediately releases the memory. On the other hand, when executed for memory acquired from the heap area, free(3) does not immediately release the memory but instead pools it, unless it is in a contiguous free area of a size equal to or greater than MALLOC_TRIM_THRESHOLD_ at the top of the heap area. The pooled memory is reusable. If this setting value is large, the maximum memory size managed in the heap area increases, reuse of memory areas may be promoted, and memory acquisition performance may improve. (However, memory use efficiency may decrease.)
MALLOC_TRIM_THRESHOLD_	Integer between 0 and ULONG_MAX <in bytes> [Decimal or hexadecimal number]	134217728 (= 128 MiB)	The setting is the threshold for immediately releasing memory when free(3) is executed for memory acquired from the heap area. If it is contiguous free memory of a size equal to or greater than the specified threshold and positioned at the top of the heap area, the memory is immediately released. If it is contiguous free memory smaller than the threshold, the memory is not released but pooled instead. If this setting value is large, memory areas acquired at one time are released less frequently, reuse of memory areas may be promoted, and memory acquisition performance may improve. (However, memory use efficiency may decrease.)

Example of checking the effect of tuning with the environment variable XOS_MMM_L_ARENA_FREE

- Application program description

  1. Acquire (malloc(3)) and release (free(3)) memory 1,024 times from a heap area with a size of 8 MiB.
  2. Execute the processing of 1 in two loops.
  3. Measure the time taken by malloc(3)/free(3) each time.

- Application program

```
#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;
}
```

- Tuning method

  Using the following two patterns, set the environment variable XOS_MMM_L_ARENA_FREE, and compare performance:

  1. Setting for immediately releasing the memory pages that can be released at the free(3) time

     export XOS_MMM_L_ARENA_FREE=1

  2. Setting for not immediately releasing the memory pages that can be released at the free(3) time

     export XOS_MMM_L_ARENA_FREE=2

- Explanation and performance prediction

  If XOS_MMM_L_ARENA_FREE=1 is set, the reserved memory area in the mmapped chunk is released immediately after free(3) is executed for the first time. Later, when malloc(3)is executed for the second time, its processing is considered to take nearly the same length of time as the processing of the first malloc(3) execution since a memory area has to be secured in the mmapped chunk again.

  If XOS_MMM_L_ARENA_FREE=2 is set, the reserved memory area in the heap area is not released after free(3) is executed for the first time. The processing of the second malloc(3) execution is considered to take a shorter length of time since the reserved memory area in the heap area is reused, reducing the memory allocation cost.

Example of checking the effect of tuning with the environment variable XOS_MMM_L_ARENA_LOCK_TYPE

- Application program description

  1. Create multiple threads, and allocate each thread to a different CPU.
     This program explicitly specifies allocated CPU numbers with consideration of the FX server configuration. For details on the CPU numbers used by application programs on the FX server, see "3.2.2 Job Memory Division Function."

  2. Issue malloc(3) individually for each thread, and secure the heap area.

  3. Execute the processing of 1 and 2 in 10 loops, and measure the time taken by malloc(3)/free(3) each time.

- Application program

```
#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;
}
```

- **Tuning method**

  Using the following two patterns, set the environment variable XOS_MMM_L_ARENA_LOCK_TYPE, and compare performance:

  1. Prioritizing memory acquisition performance (processing dynamic memory reservation requests in parallel)

     export XOS_MMM_L_ARENA_LOCK_TYPE=0

  2. Prioritizing memory use efficiency (processing dynamic memory reservation requests sequentially)

     export XOS_MMM_L_ARENA_LOCK_TYPE=1

  The environment variable XOS_MMM_L_ARENA_FREE=2 is also set with each pattern. This facilitates processing cost comparison in cases of contention among dynamic memory reservation requests by multiple threads. Also, the resulting processing cost for physical page allocation is low.

  The above example program gets 20480 times of memory of size 64 KiB per 1 thread. It competes in 16 threads and measures 10 times.

- **Explanation and performance prediction**

  The possibility of contention in memory acquisition processing with malloc(3) is high when multiple threads issue malloc(3) simultaneously.

  If XOS_MMM_L_ARENA_LOCK_TYPE=0 is set, a thread heap area is generated when malloc(3) contention occurs, thereby enabling dynamic memory reservation requests to be processed in parallel. This is considered to shorten the time taken for memory acquisition.

  IF XOS_MMM_L_ARENA_LOCK_TYPE=1 is set, a thread heap area is not generated when malloc(3) contention occurs. Also, threads share the process heap area. For this reason, their processing must be done sequentially while waiting for the malloc(3) processing currently in progress to complete. This is considered to lengthen the time taken for memory acquisition.

3.5 Debugging an Application Program

This section presents useful open source software (OSS) for memory allocation debugging and tuning for application programs.

- Valgrind

  Valgrind is a dynamic analysis tool for detecting memory leaks and thread errors.

  Valgrind consists of functions such as memcheck, cachegrind, callgrind, and helgrind.

  You can use the cachegrind and callgrind functions simultaneously with the large page library. The other functions hook malloc, so they cannot be used simultaneously with the large page library.

  Note

  In the execution of a job using helgrind to debug an application program, OOM (Out of memory) killer may kill the helgrind process.

  According to specifications, helgrind references the entries corresponding to the data seg size (RLIMIT_DATA) and stack size (RLIMIT_STACK) in resource limit values at the process start time, and tries to secure virtual memory for them. helgrind creates PTEs

(Page Table Entries) in system memory to configure a virtual memory space. However, if either of the aforementioned resource limit values is set to unlimited, helgrind tries to secure unlimited virtual memory and creates a PTE that exceeds the allowed amount of system memory. Consequently, OOM occurs.

To prevent this event, specify appropriate resource limit values in the proc-data and proc-stack options of the pjsub command when executing the job. The options correspond to the data seg size (RLIMIT_DATA) and stack size (RLIMIT_STACK), respectively. For details on how to specify resources when executing a job, see the descriptions about specifying resources in "Options at the job submission time" in the "Job Operation Software End-user's Guide" manual. You can check limit information, such as system resource limit values, with the pjacl command. See "Checking restriction information" in the "Job Operation Software End-user's Guide" manual.

Information

For details on Valgrind, see the Valgrind OSS community site information (http://valgrind.org/) and the RHEL developer's guide information (https://access.redhat.com/documentation/).

- malloc statistical information

  malloc statistical information (malloc_stats(3)) is a glibc function for outputting the amount of memory allocated to each area.

  To use malloc statistical information in glibc, you need to modify (add a function call) and rebuild the source code.

Information

For details on glibc, see the glibc OSS community site information (https://www.gnu.org/software/libc/).

Appendix A Messages

This appendix describes the messages output by the command (fhetbo command) of the HPC tag address override control function and by the large page library. The messages appear at the standard error output.

A.1 HPC Tag Address Override Control Function (fhetbo Command)

A.1.1 How to Read Messages

The messages output by the HPC tag address override control function (fhetbo command) of the Fujitsu HPC extension function have the following format.

```
message-type component-name - message-text
```

The following table shows the components of a message from the Fujitsu HPC extension function.

Table A.1 Message Components

Component	Meaning
message-type	Displays one of the following message types: [INFO] Information message
component-name	Displays "the xos FHE number". "number" is a unique message identification number.
'-'	Delimiter character
message-text	Displays the details of the event that occurred.

A.1.2 Messages

Information Message

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

Meaning

Tag address override is enabled for the core mask shown by coremask.

jobid: Job ID

coremask: Mask value of the core where tag address override is enabled

Effective bits are set to 1 in the order of core number starting from 0 from the least significant bit.

For example, if coremask is "0xfffffffffffffffff000", this means that tag address override for core 12 - 59 is enabled.

Action

No action is necessary.

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

Meaning

Tag address override is disabled for the core mask shown by coremask.

jobid: Job ID

coremask: Mask value of the core where tag address override is disabled

Disabled bits are set to 1 in the order of core numbers starting from 0 from the least significant bit.

For example, if coremask is "0xfffffffffffffffff000", this means that tag address overrides for core 12 - 59 is disabled.

Action

No action is necessary.

A.2 Large Page Library

A.2.1 How to Read Messages

The messages output by the large page library have the following format.

```
message-type component-name - message-text
```

The following table shows the components of a message from the large page library.

Table A.2 Message Components

Component	Meaning
message-type	Displays the following message type: [WARN] Warning message
component-name	Displays "the xos LPG number". "number" is a unique message identification number.
'-'	Delimiter character
message-text	Displays the details of the event that occurred.

A.2.2 Messages

Warning Message

[WARN] xos LPG 2001 - Failed to allocate HugeTLBfs pages, going to try allocating normal pages.

Meaning

Normal pages were acquired because acquisition of large pages (HugeTLBfs) failed.

Action

The possible factors in the failure to acquire large pages include insufficient free memory and fragmented memory. Check the amount of memory used by the job and the amount of free memory. As needed, ask the system administrator about the amount of memory available to a job. If this phenomenon recurs despite an appropriate amount of memory used by the application program and appropriate system settings, collect a core dump and contact a Fujitsu systems engineer (SE) or Fujitsu Support Desk.

[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.

Meaning

The .data/.bss area cannot be on a large page (HugeTLBfs) because the binary format of the application program is a PIE (Position Independent Executable). Using a normal page for the .data/.bss area, execution continued.

a.out: Application program

Action

To put the .data/.bss area on a large page, compile the application program while preventing the binary format of the program from becoming PIE. (Using the readelf -h command, create the application program in such a way that e_type becomes ET_EXEC).

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

Meaning

The .data/.bss area for the application program cannot be on a large page (HugeTLBfs) because the address range of the .data/.bss area conflicts with another segment. Using a normal page for the .data/.bss area, execution continued.

seg_index1, seg_index2: Segment index value

Action

To put the .data/.bss area on a large page, specify an appropriate linker script (linker option) for the application program compile time to align the virtual address of the .data/.bss area to the large page size.

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

Meaning

The thread stack cannot be on a large page (HugeTLBfs) because the virtual address of the thread stack is specified in the second argument attr (attribute) of pthread_create(3). Using a normal page for the thread stack, execution continued.

stack_addr: Specified stack address

stack_size: Specified stack size

Action

To put the thread stack on a large page, do not specify the virtual address of the thread stack in the second argument attr (attribute) of pthread_create(3).

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

Meaning

The process stack cannot be on a large page (HugeTLBfs) because multiple threads were detected during the processing to put the process stack on a large page. Using a normal page for the process stack, execution continued.

Action

There must be only a single thread during the processing to put the process stack on a large page. Check whether your program generates a thread. Also consider that other libraries, not just the large page library, can also generate threads.

[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*

Meaning

The process stack cannot be on a large page (HugeTLBfs) because the process stack conflicts with another memory area, which is located after (at a higher address than) the stack. Using a normal page for the process stack, execution continued.

stack_bottom_addr: Bottom address of the stack

next_vma_addr: Start address of the memory area after the stack

Action

To put the process stack on a large page, the address range must be guaranteed not to overlap with another memory area. Check whether your program explicitly uses mmap(2) to map a memory area near the process stack. Also consider that dynamic libraries other than the large page library can also map memory areas.

[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*

Meaning

The process stack cannot be on a large page (HugeTLBfs) because the process stack conflicts with another memory area, which is located before (at a lower address than) the stack. Using a normal page for the process stack, execution continued.

stack_top_addr: Start address of the stack

prev_vma_addr: Bottom address of the memory area before the stack

Action

To put the process stack on a large page, the address range must be guaranteed not to overlap with another memory area. Check whether your program explicitly uses mmap(2) to map a memory area near the process stack. Also consider that dynamic libraries other than the large page library can also map memory areas.