Optimized Lightweight Thread Framework for Mobile Devices

Geunsik Lim*, Changwoo Min†, Sang-Bum Suh‡, Hyun-Jin Choi§ and Young Ik Eom¶
*†¶Sungkyunkwan University, Korea
*†‡§Samsung Electronics, Korea
Email: {leemgs*, multics69†, yieom¶}@ece.skku.ac.kr, {sbuk.suh‡, hj89.choi§}@samsung.com

***Abstract*—One of the main changes in the current Linux is that the Linux thread model is transferred from an existing thread model to Native POSIX Thread Library (NPTL) for scalability and high performance. Each user-space thread is implemented as a corresponding kernel thread for fast creation and termination; 1:1 mapping model. Multiple threads in a single process can make better use of multiple processor cores. Since a user-level thread is implemented as a corresponding kernel thread, it is individually schedulable and manageable. Each thread in a multi-processor system will be able to run simultaneously in different CPU.**

NPTL in Linux 2.6 improves scalability and performance of server and desktop over Linux 2.4. But, it is inadequate on embedded systems such as mobile phone and DTV, since embedded systems have limited physical resources including CPU clock-speed and memory capacity.

In this paper, we introduce a lightweight thread framework to enhance NPTL on GLIBC/EGLIBC for embedded devices. Our solution consists of (1) stack management to reduce memory footprint, (2) thread scheduling to improve responsiveness, and (3) developer support for debugging and profiling. These approaches provide a cost effective development opportunity to the embedded developers of commercial mobile devices.

***Index Terms*—Lightweight process (LWP); Thread model; Thread scheduling; Thread stack; Thread naming**

I. Introduction

Most embedded systems such as mobile phone, camcorder, and digital TV (DTV) have been designed for specific purposes. As customers expectation on embedded devices gets higher, manufacturers provide richer functionalities to raise their competiveness. One of essential software techniques to provide richer functionalities is supporting more number of concurrent threads. The number of concurrent threads running on recently released embedded devices such as camcorder and mobile phone is ranging from 200 to 700.

Moreover, since many embedded devices recently released allow a user to download applications from app-store directly and install them after purchasing, the number of concurrent threads or processes is going to be larger. Thus, supporting larger number of concurrent threads and processes in a cost effective manner is critically important. [1]

Obvious solution to support more concurrent threads is to use a faster CPU and larger memory. However, it increases manufacturing cost and potentially decreases the competiveness of product in the consumer market. Linux kernel adopts Native POSIX Thread Library (NPTL) for multiple thread support from version 2.6. NPTL is specially designed for scalability and performance by using 1:1 mapping between user level thread and kernel thread. But, since it is designed for servers and desktops, there are impedance mismatch to embedded systems with limited CPU and memory.

Mobile embedded systems provide limited physical resources due to low power management and cost competitiveness. Moreover, a swap device to overcome physical memory shortage is not supported in most embedded environments. Therefore, application developers should obey good thread programming styles and find an optimal stack size for threads to implement efficient applications. In this way, we can implement an efficient and economical system without additional hardware support.

In this paper, we introduced optimized NPTL framework for resource constraint mobile devices. Our solution consists of (1) stack management to reduce memory footprint, (2) thread scheduling to improve responsiveness, and (3) developer support for debugging and profiling.

The rest of this paper is organized as follows. Section II describes the design and implementation of the proposed schemes. Section III shows the evaluation results of the schemes. Related work is described in Section IV. Finally, in Section V, we conclude the paper.

II. Design and Implementation

In this section, we describe the design and implementation of our proposed schemes. Figure 1 shows the overall architecture of the optimized lightweight thread framework. We extend NPTL framework by adding gray colored components in Figure 1. When a user-space thread is created, the *thread naming* component monitors a newly created thread to find out the purpose of each thread for optimization and debug support and fixing bugs. The *stack management* component allocates suitable stack size [2] for each thread to minimize memory footprint. The *thread profiling* component provides the profiling information including stack size, guard size, scheduling priority, and time gap to optimize boot-time and resource usage. Finally, the *thread scheduling* component controls dynamic scheduling of threads to reduce user-perceived latency by using our proposed *Task scheduling importance hierarchy*.

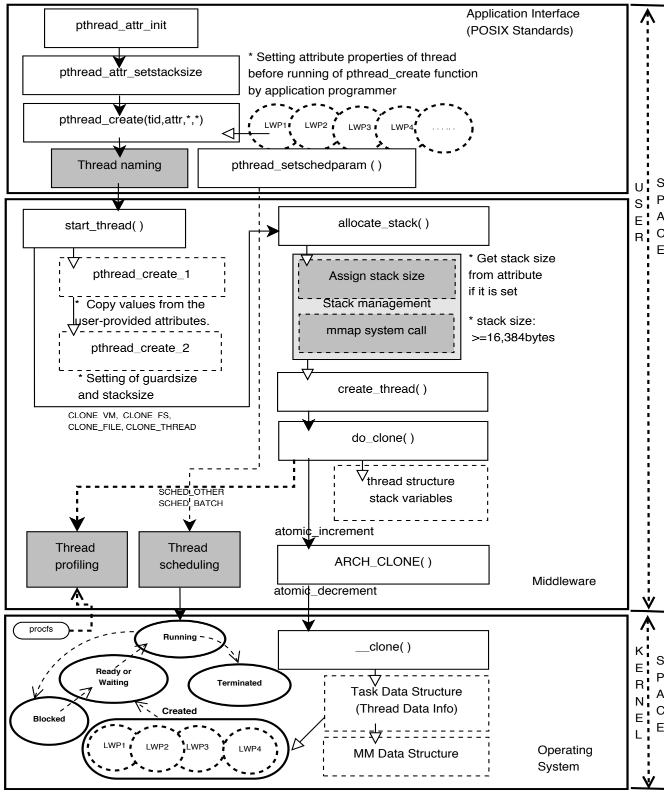

Fig. 1. Overall architecture of the system

A. Stack management

One main reason of a *segmentation fault* caused by user-space thread libraries is too low memory allocation. If we try to allocate too much memory, the operating system also consumes too much memory, and thus it leads to *segmentation fault* error. In many cases, we can avoid the *segmentation fault* error by adjusting the stack to suitable size. Except for expansion of physical memory and using swap space, there are two software approaches to adjust the stack size:

- Changing system wide default stack size by using `ulimit` command which is usually incorporated with booting process: We have to decide the suitable stack size of the embedded system, which wants to be selected as a default stack size value for all threads that are created in the specified embedded system. Through it, we can manage the policy consistently that adjusts the default stack size of the thread at the middleware level. But, this approach has the one problem that can not manage a suitable stack size of each thread in detail.
- Specifying stack size of an individual thread by using POSIX API (`pthread_attr_`): Although, this approach controls each thread by adjusting manually this POSIX API to each thread, we have the one issue that can not still manage a lot of user-space threads automatically and effectively.

To solve the unresolved issues of above two software approaches, we propose an appropriate stack size policy using the *stack management* component to manage all user-space threads automatically in the NPTL layer.

First of all, the existing NPTL creates a stack size of 8 MB as a default of a user-space thread. But, this setting is suitable for enterprise server environment. According to our profiling, the required thread stack size for most embedded application is less than 1 MB. Considering the minimum memory space of the data structure for the creation of threads, the Linux-based embedded system needs the stack size more than 16 KB per a thread essentially.

We calculate a maximum stack size used by profiling all user-space threads via the data structure of memory management to decide the default stack size in the embedded system. The application developer does not need to know anything about a stack related knowledge, because the *stack management* component automatically decides the stack size of newly created user-space thread with `pthread_create()` library call.

B. Thread naming

When a system crash is happened and we need to execute performance optimization among user-space threads, figuring out a essential role of a user-space thread is very important. But, it is difficult that we try to measure a thread's role with only PIDs, because there are more than hundreds of user-space threads are concurrently running in the modern embedded systems.

It is possible to distinguish the main role of relevant child threads by using the name of the thread function set as the third parameter from `pthread_create()`.

Otherwise, the application developer only gets a unique value of each thread with Thread Local Storage (TLS) [3] that is supported by CPU and cross-compiler. This value is used for the purpose of identifying what thread runs a specified function at some point. However, when hundreds of threads call the same function by using the `pthread_create()` function, it is not easy to know the unique purpose of each thread that is executed by this method.

We extended interface which is called *thread naming* in the user-space thread model based on NPTL to solve these problems. It would be easy to understand the operation purpose of all threads in the platform.

Because, many teams create many threads to develop a lot of packages in case of a large scale project, it often causes nonproductive activities to understand the operation purpose of threads each team produces.

We implemented thread interface like `pthread_set_naming_np()` library call and `pthread_get_naming_np()` library call additionally for the our embedded system. The *thread naming* component supports the detailed information of all threads produced by defining the role of the additional thread using the `pthread_set_naming_np()` library call.

C. Thread profiling

We decide the optimal moment for booting the system by profiling the time interval of thread creations between the front

and the back, and a CPU sharing of all threads created before the GUI initial screen of the embedded platform appears.

When the CPU utilization of a specific moment is low during the system booting, we can maximize the CPU utilization and shorten the system booting time by parallelizing independent functions of threads. And, the generation of threads happened after a specific time from a specific thread's execution that analyzes CPU utilization of threads executed for a long time in detail. If CPU utilization was not high, it means there is room for optimization. Although CPU utilization is high during the embedded system booting, and if the relevant scheduling work could be possible after an initial screen appears, it would be effective to run those threads after the initial screen' appearance.

D. Thread scheduling to reduce user waiting time

1) Extension of `pthread_{set|get}_priority_np` *interface:* When Linux based on 3.0 version uses system call and library call, it consists of a total of 140 priorities with normal priority level using nice value from -20 to 19 and real-time priority level from 1 $\sim$ 99. Low numbers have high priority in Linux kernel-space.

Normal priority is defined in the file of `sched.c` and Linux schedule this tasks with O(1) scheduler or CFS scheduler [4] depending on Linux version, after allocating one normal priority between bitmap 100 and bitmap 139 about a nice value between -20 and 19 by user-space application developers.

User-space real-time support and a few challenges for 100% POSIX compliance were written in "Native POSIX Threads Library for Linux" [5] paper by Ulrich Drepper.

Infrastructure for POSIX compatible user-space real-time support was improved by adding the features like Priority Queuing, Robust Mutex(=RT MUTEX) [6] and Priority Inheritance [7] [8]. This means application developers can realize the real-time thread programming in user-space. Table I below shows the system call and library call for setting the scheduling priority against the process/thread with normal priority and real-time priority, respectively.

Scheduling Priority	*PID TID*	*Function Name* (API)	interface	
			LinuxThread	*NPTL*
Normal (-20 to 19) NON-ROOT	PID	setpriority() nice()	getpid()	gettid()
	TID	setpriority() nice()	getpid()	gettid()
Real-time (1 to 99) ROOT	PID	sched_setscheduler() sched_setparam()	getpid()	gettid()
	TID	pthread_setschedprio() pthread_setschedparam()	getpid()	gettid()

TABLE I
LINUXTHREAD VS. NPTL SCHEDULING SYSTEM CALL COMPARISON

The scheduling priority of an already-running normal priority thread can be changed by calling a system call like `setpriority()`, `nice()`.

In case of tasks having a real-time priority value, there are possible values for scheduling policy like SCHED_RR (real-time round-robin policy), SCHED_FIFO (real-time FIFO policy), SCHED_OTHER (for regular non-real-time scheduling) and so on. A parameter pointer showing the scheduling priority in user-space can set the priority order ranging from 1 to 99 for the purpose of real-time scheduling policy. The priority of threads using SCHED_BATCH for 'batch' style execution is counted as 0.

Considering real-time property under embedded environment SCHED_RR seems ideal, SCHED_FIFO is more useful to take effect of performance practically because a simple policy is good for performance and effective management.

```
struct sched_param { int sched_priority; };
```

Because the use of `gettid()` depends on each CPU architecture in Linux 3.0, system call number is different among CPU architectures. We utilized `gettid()` system call normally after defining manually like the method below because of non-implementation of `gettid()` in Linux system.

```
/* Appending gettid syscall in user-space */
#define gettid() syscall(__NR_gettid)
```

The unique number of thread executed in the related function region to apply normal priority to threads that are created as nice value. The `gettid()` function has to be made using `syscall(__NR_gettid)`. And then, the use of `gettid()` function is available to utilize the `gettid()` function by `syscall(__NR_gettid)` in the function of relevant thread. We use `gettid()` instead of `getpid()` in the NPTL thread model to find out this thread. Above `syscall()` function returns kernel-space thread id that mapped about user-space thread id that is running by including `unistd.h` header file. The `gettid()` system call is defined as below in the file of `timer.c`

```
/* gettid syscall details in Linux */
asmlinkage long sys_gettid(void){
  return current->pid;
}
```

When you try to utilize the `gettid()` system call using the above method, it is recommended to add the thread library function including system calls by considering the impact of embedded system's performance because of the cost of system calls. It is very useful to measure the execution time, calls and errors for system calls of thread library function to be added to know the cost of CPU utilization. The `call.S` file of the ARM Architecture defines `sys_set_thread_area()` as `sys_ni_syscall (224)` and `sys_get_thread_area` as `sys_ni_syscall (225)`.

Maintenance of source code of a large scale project can go on smoothly by not mixing many different functions preferred by developer in embedded platform but rather using a uniform common interface. We extended the thread function of `pthread_set_priority_np()` or `pthread_get_priroity_np()` additionally for the application developer to get ID value of a thread easily.

2) Controlling CPU scheduling of user-space thread: By increasing the speed of user's application under embedded system environment at specific time, users often want to get shorten application's waiting time. The support of these mechanisms raises the flexibility of scheduling priority for CPU utilization when threads need higher CPU utilization at specific time. Effective throughput of applications is possible by grouping thread applications based on the importance of processing speed and response speed in embedded system having limited CPU performance.

We need the thread dealing mechanism to realize the way to give a suitable scheduling priority value of thread. So, we newly designed *Task scheduling importance hierarchy*. Table II below gives an explanation about task classification and task meaning according to *Task scheduling importance hierarchy* table. We can minimize user's waiting time for embedded devices by adjusting the self thread's scheduling priority at specific time or by changing other thread's normal priority at run-time dynamically with `pthread_setschedparam()` library call in Linux 3.0 based NPTL environment.

Hierarchy of sched priority	Description
Busy Task (Urgent)	Busy task means the threads in the top of screen which interact with user or which occupy CPU utilization under processing CPU.
Foreground Task (Normal)	Foreground task is thread that appear in the screenof user's embedded device but doesn't have activity to be processed immediately.
Service Task (Support)	Service task is middleware level component which supply important functions for processing of application and thread that occupies service.
Background Task (Hidden)	Background task is thread that occupies activity not visible to user.
Idle Task (Unlimited)	Idle task is thread that doesn't occupy component of any active application in embedded system.

TABLE II
TASK SCHEDULING IMPORTANCE HIERARCHY

The pseudo code below shows the implementation of NPTL library to control the scheduling priority arbitrarily or by force for user-space threads based on normal priority that are created on non-preemptive Linux 3.0.

```
__pthread_setschedparam(tid,policy,param)
{/* Normal(=dynamic) priority for O(1) / CFS */
struct pthread *pd=(struct pthread *)tid;
if (policy==SCHED_OTHER || policy==SCHED_BATCH){
/* Scheduling priority of thread */
int which = PRIO_PROCESS ;
/* Handling of SCHED_OTHER priority */
if ( param->sched_priority < -20 &&
     param->sched_priority > 19 )
     return nice_range_error;
if (nice_gap < 5 && policy == SCHED_BATCH)
    cfs_aware_manager(nice_gap);/* for cfs env */
/* Getting LWP(thread id of kernel) to
 * change scheduling priority about tid */
if (setpriority(which,unique_kernel_tid(),
    param->sched_priority) ){
   perror("setpriority() operaton error.\n");
   result = errno;
}}
```

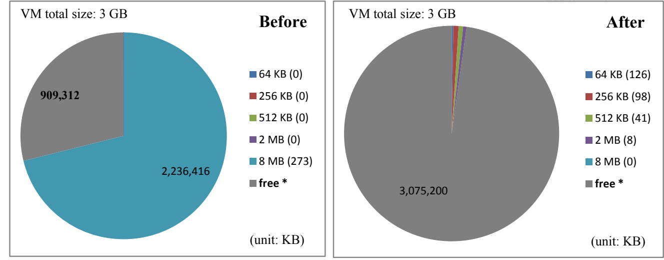

Fig. 2. The stack size of user-space threads

As mentioned above, after improving scheduling-related thread function of NPTL library, The way described below can control the thread application's scheduling actively to apply different scheduling priority to many threads which are created in one process in embedded system.

```
/* Aggressive Thread scheduling for
 * urgent threads arbitrarily & by force */
struct sched_param schedp;
/* priority number of between -20 ~ 19. */
int priority = -20 ;
memset(&schedp, 0, sizeof(schedp));
schedp.sched_priority = priority;
/* for controlling self thread */
pthread_setschedparam(pthread_self(),
SCHED_OTHER/SCHED_BATCH, &schedp)
/* for controlling another thread */
pthread_setschedparam(thread[i],
SCHED_OTHER/SCHED_BATCH, &schedp)
```

III. EVALUATION

We introduced several approaches for improving the current NPTL thread model: a suitable thread stack size for embedded environments, thread naming interface expansion for optimization, supports of thread profiling and debugging components to minimize the boot time of embedded platform, a thread priority management method according to scheduling importance of thread application, an arbitrary or enforced thread scheduling control policy to speed up user application processing.

From our experimental results, Figure 2 shows that we got a fewer memory footprint via stack-size enhancement of the existing user-space thread model on EGLIBC and GLIBC. The 'free' word in Figure 2 means available virtual memory for user-space applications. Actually, we saved virtual memory resource innovatively against the existing NPTL model on our embedded system that is running 273 threads.

As a result of that, We are keeping a suitable memory size without extension of physical memory for the heavy-weight threads based embedded platform according to the increased the number of threads.

Table III shows the information like stack size, guard size, priority value, and time-gap to optimize the system booting time. Performance optimization can be possible by debugging internal operation information of user-space functions and

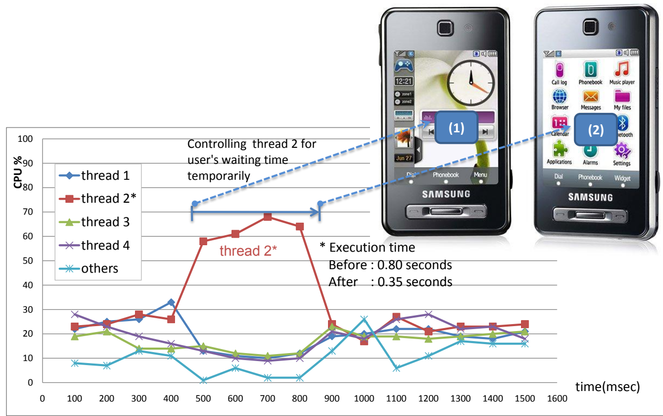

Fig. 3. Mobile device after adjusting the lightweight thread framework

identifying naming information of relevant threads because the thread number 168 is executed almost for 1 second in Table III below. Additionally, readability can be improved by understanding the detailed interactions among threads to analyze the purpose of creating, blocking, sleeping, and finishing each thread.

Name (process)	*TID* (thread)	*StackSZ* (kbyte)	*Priority* (nice)	*Time Gap* (msec)
Files Copy Extension	162	256	5	195
LifeCycle Controller	163	256	0	3
Micom Task	164	256	0	128
Event Dispatcher	165	256	5	115
Node Manager	166	256	5	2
Task Extension	167	256	5	2
Media API	168	256	5	977
Msg Event Handler	169	256	10	3
-	-	...	-	-

TABLE III
THREAD NAMING AND PROFILING RESULT

We minimized user's waiting time by controlling the thread's dynamic priority at run-time instantly with `pthread_setschedparam()` library based on *Task scheduling importance hierarchy* whenever users pushes a menu to run a specific software. Figure 3 shows that the improved NPTL thread model has better performance by reducing user's waiting time from 0.80 seconds to 0.35 seconds on CFS scheduler in our experiments.

IV. Related Work

The existing NPTL library by Ulrich Drepper [5] is mainly designed for scalability and performance on enterprise server environment.

Wheeler [9] proposed the qthread API to solve problems that must be addressed because massive parallelism is to be popularized in NPTL thread model. It provides basic lightweight thread control and synchronization primitives in a way that is portable to existing highly parallel architectures. But, this approach is also designed for server environment only.

There are "NPTL Stabilization Project" [10] and "Native POSIX Threads Library Support for uClibc" [11] for utilizing NPTL on embedded systems. "NPTL Stabilization Project" does not describe optimization for a lightweight embedded mobile environment. "Native POSIX Threads Library Support for uClibc" described a small memory footprint, but he does not consider scheduling, memory, and productivity synthetically like this paper.

This paper tackles these issues and proposes many ways of achieving the improvements.

V. Conclusion

The property of embedded system environment has limited physical condition like low CPU clock speed and small memory size. Therefore, the existing embedded systems using NPTL need to be improved by operating lightly and speedily with the best technical methods.

Our results confirm that the existing NPTL thread model in Linux based on O(1)/CFS scheduler can be utilized for embedded system through several improvement features with both GLIBC and EGLIBC. We reduce user's waiting time from 0.80 seconds to 0.35 seconds without any *segmentation fault* errors according to our solutions that consists of (1) stack management to reduce memory footprint, (2) thread scheduling to improve responsiveness, and (3) developer support for debugging and profiling. Our contributions provide a cost effective development opportunity for embedded developers to start developing the thread models for embedded systems through existing open sources like Linux.

Acknowledgment

We would like to thank HyoYoung Kim and Deukhyeon An for their valuable review comments. This work was supported by the IT R&D program of MKE/KEIT [10041244, SmartTV 2.0 Software Platform].

References

[1] U. Drepper, "What Every Programmer Should Know About Memory," in *Redhat*, 2007.
[2] G. Lim, "NPTL Optimization for Lightweight Embedded Devices," in *Ottawa Linux Symposium*, 2011.
[3] U. Drepper, "[TLS]ELF Handler For Thread-Local Storage," in *Redhat*, 2003.
[4] S. Rostedt, "CFS Scheduler Design," in *Linux kernel documentation*, 2007.
[5] U. Drepper, "Native Posix Thread Library for Linux," in *Ottawa Linux Symposium*, 2003.
[6] S. Rostedt, "RT-mutex Subsystem with PI Support," in *Linux kernel documentation*, 2004.
[7] I. Molnar, "PI-futex," in *http://lwn.net/Articles/177111/*, 2006.
[8] U. Drepper, "Futexes Are Tricky," in *Ottawa Linux Symposium*, 2004.
[9] K. Wheeler, R. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *IEEE International Symposium on Parallel and Distributed Processing, 2008.*, pp. 1 –8, April 2008.
[10] S. DECUGIS, "NPTL Stabilization Project (NPTL Tests and Trace)," in *Ottawa Linux Symposium*, 2005.
[11] S. J. Hill, "Native POSIX Threads Library (NPTL) Support for uClibc," in *Ottawa Linux Symposium*, 2006.