TOSHIBA
Leading Innovation

CE Linux Forum

Recent new features in NPTL

NPTL: Native POSIX Thread Library

(株)東芝　コアテクノロジセンター
エンベディッドシステムコア技術開発部
熊谷宏樹
2007年 8月 31日

はじめに

- **昨年の CELF Jamboree #10 では MIPS での基本性能 について、NPTL と linuxthreads を比較して御紹介。**

- **今回は、ここ1年くらいで使用可能になった、NPTLの機能を御紹介。**

ここ1年の動向

	Linux kernel	glibc / NPTL
2006/6	2.6.17 Robust Mutex (futex)	
2006/9	2.6.18 Priority Inheritance Mutex (futex)	2.5 Robust Mutex Priority Inheritance Mutex Priority Protect mutex
2006/11	2.6.19	
2007/2	2.6.20	
2007/4	2.6.21	
2007/5		2.6
2007/7	2.6.22 fix "futex priority based wakeup" (non-PI)	2.5.1 / 2.6.1 fix "robust mutex does not work if owner died with multiple waiters" bugzilla #4512

今回、話題に取り上げる機能

紹介内容

略称

PI = Priority Inheritance
PP = Priority Protect

1. **Priority (Inheritance/Protect) Mutex**
   - **ユーザ空間スレッドの優先度逆転問題を解決。**
     → 高優先度スレッドの応答性の向上が期待できる。

← 実装比較及び性能評価について報告。

2. **Robust Mutex**
   - **消滅したコンテキストが持っていたミューテックスを回復できる。**
     → プロセスとの同期を行うシステム構築に適用できる。

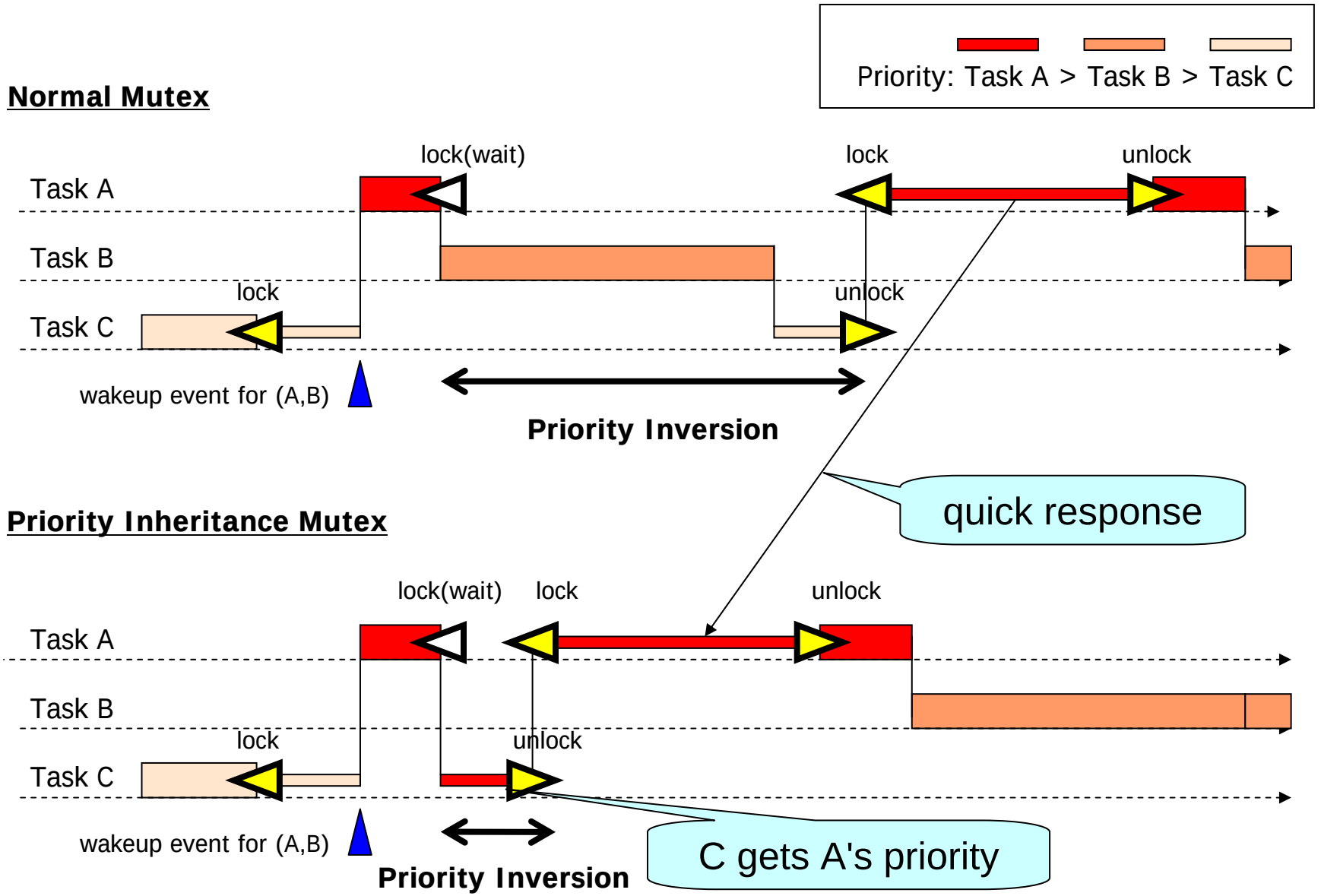
← プロセス間同期の評価について報告。

Priority (Inheritance / Protect) Mutex

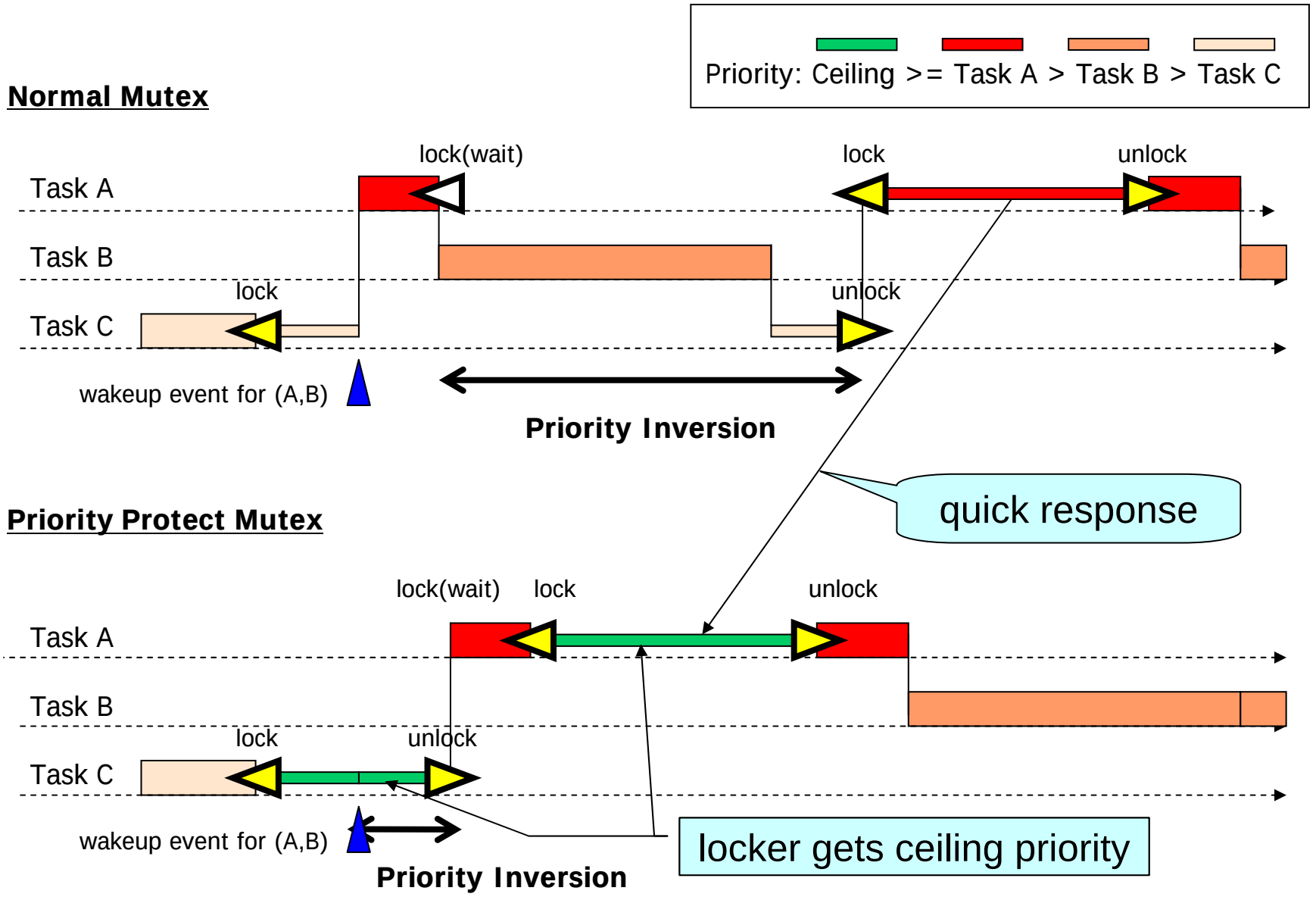
Priority (Inheritance / Protect) Mutex

	Priority Inheritance Mutex	Priority Protect Mutex
動作	ロックを持っているスレッドは、ロックを待っているスレッド※の優先度で実行する。 ※ 高い優先度を持つ場合。	ロックを持ったスレッドは、設定されたシーリング優先度※で実行する。 ※ロックネスト時は、一番高い優先度。
特徴	優先度逆転をバウンド可能。 ロックが競合しない限り、優先度の変更は必要なし。	優先度逆転をバウンド可能。 優先度の変更処理が定常コストに。 シーリング優先度の設定値決定が困難。
サポート	kernel 2.6.18 ~ glibc 2.5 ~	glibc 2.5 ~
注意	1. Non-PI Mutex (非優先度継承)で、起床順が優先度順にならない問題。 "futex priority based wakeup" (fixed in kernel 2.6.22~) 優先度継承の実装(plist対応)が通常のfutexにも適用され修正。	1. NPTLでは、Robust Mutex と併用できない。 2. SCHED_OTHERクラスのスレッドでは使えない。

Priority Inversion (PI Mutex)



Priority Inversion (PP Mutex)

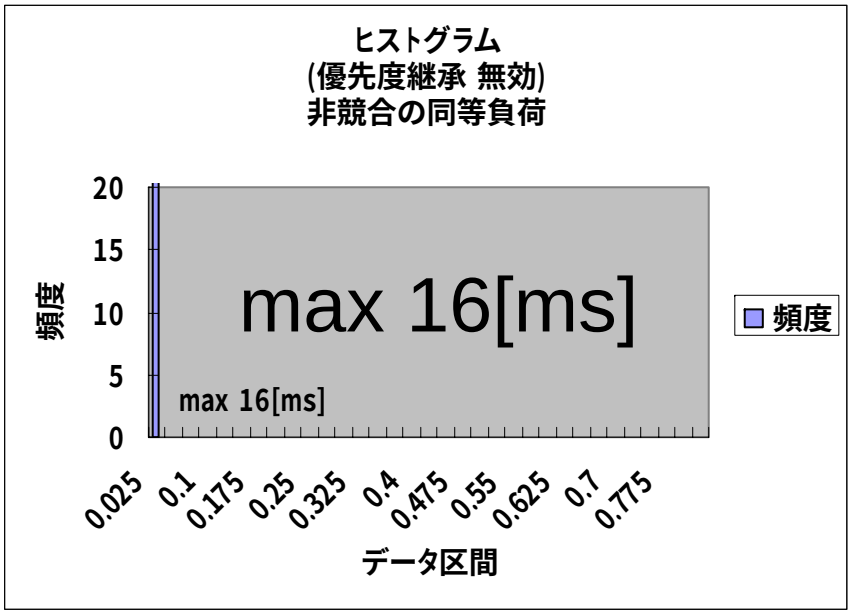


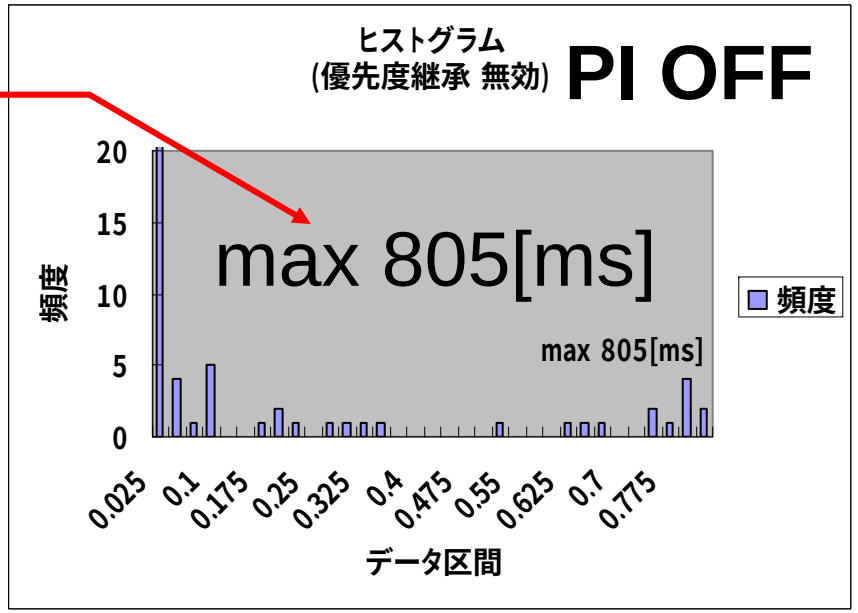
Priority Inversion Test - PI

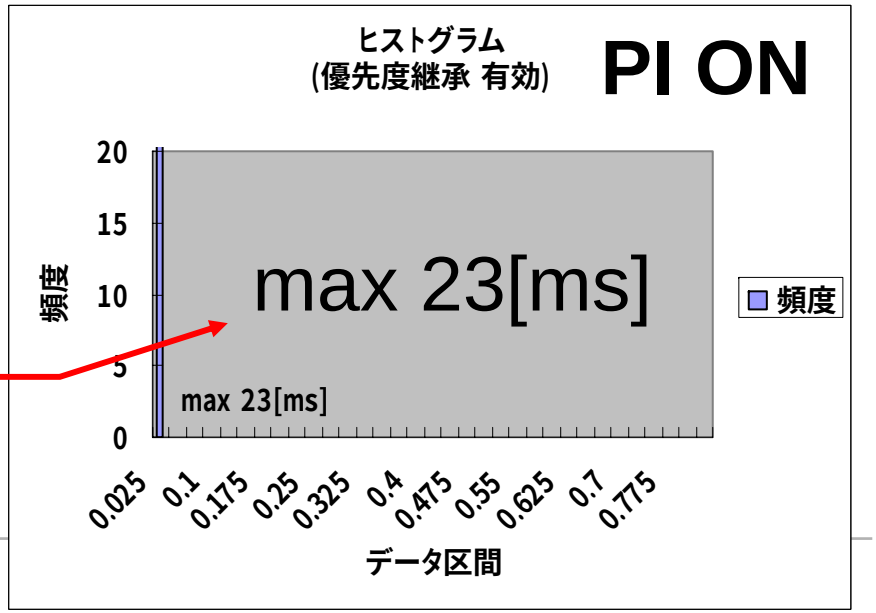
- **実システムに外乱負荷を加えて優先度逆転に対する効果をテスト。**
  - **外乱負荷**
    - **ベストエフォートクラスのタスクを追加し、メッセージキューで使用するMutexを lock/unlock し続けるタスクを追加。**
  - **測定対象**
    - **外乱を加えたメッセージキューにおける送信から受信までの処理時間（最悪値）。**

Priority Inversion Test - PI

50倍以上悪化







大きな劣化無し
(効果を確認できた)

Mutex Implementation in NPTL

- **2つの処理パターンで実装を比較。**
  - fast path
    - **ロック競合が発生しない場合**
      (no contention)
  - slow path
    - **ロック待ち＆待ち解除が必要な場合**
      (wait & wakeup)

Mutex Implementation in NPTL (fast path)

var **は ロック変数** (mutex->__data.__lock)

var = (前提とする値) 変更後の値

fast path		user (glibc)	kernel
normal mutex	lock	var = (0) 1	
	unlock	var = (1) 0	
PI mutex	lock	var = (0) TID	
	unlock	var = (TID) 0	
PP mutex	lock	sched_setscheduler var = (ceil) ceil | 1	ceiling priorityへ優先度変更
	unlock	var = (var) ceil sched_setscheduler	base priorityへ優先度変更

ユーザ空間で処理が完結

ceiling prio > thread prio **の場合、優先度変更処理が常に入る**

Mutex Implementation in NPTL (slow path)

slow path		user (glibc)	kernel
normal mutex	lock	var = (1) 2 futex(FUTEX_WAIT) var = (0) 2	schedule
	unlock	var = (*) 0 futex(FUTEX_WAKE)	wake_up
PI mutex	lock	futex(FUTEX_LOCK_PI)	var = (var) var | FUTEX_WAITERS priority boosting rt_mutex lock
	unlock	futex(FUTEX_UNLOCK_PI)	var = (var) TID | FUTEX_WAITERS rt_mutex unlock priority unboosting
PP mutex	lock	sched_setscheduler var = (ceil|1) ceil|2 futex(FUTEX_WAIT) var = (ceil) ceil|2	ceiling priorityへ優先度変更 schedule
	unlock	var = (var) ceil futex(FUTEX_WAKE) sched_setscheduler	wake_up base priorityへ優先度変更

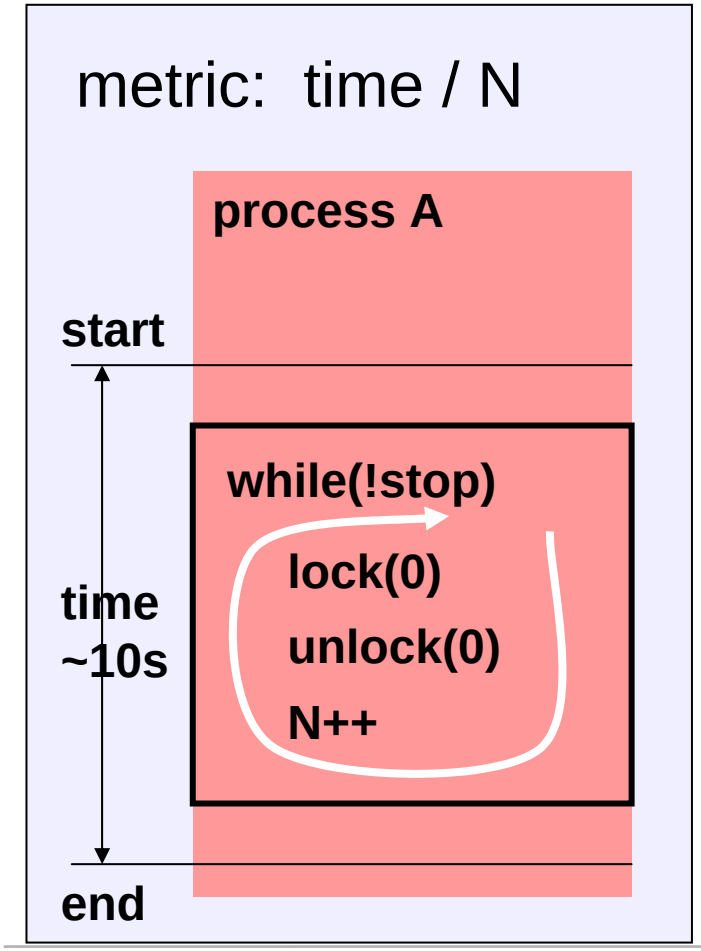
同期処理は同じfutex

(※TIDは待ち中タスク)

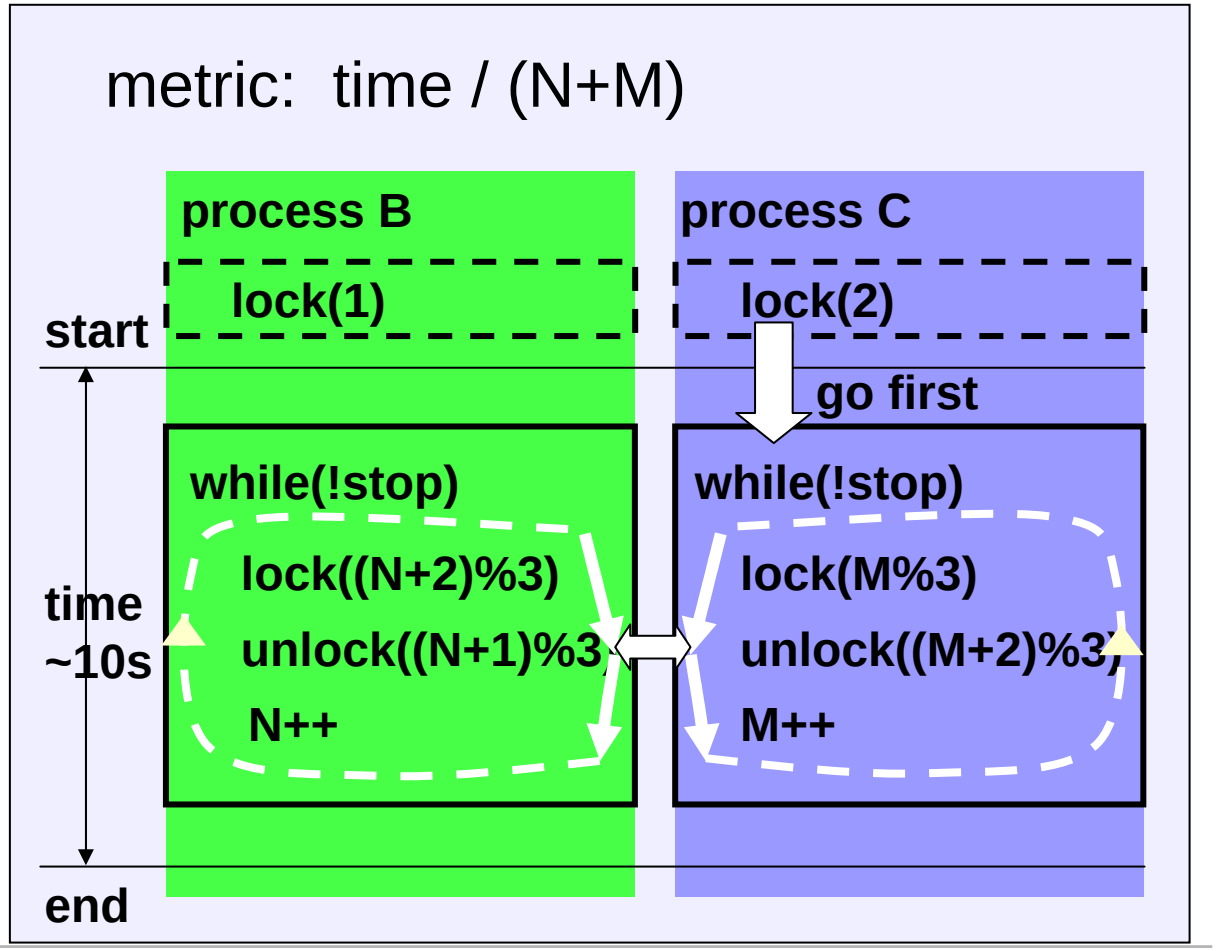
rt mutex が backend となっており、
優先度継承が行われる。

Mutex Performance

Call overhead (fast path)

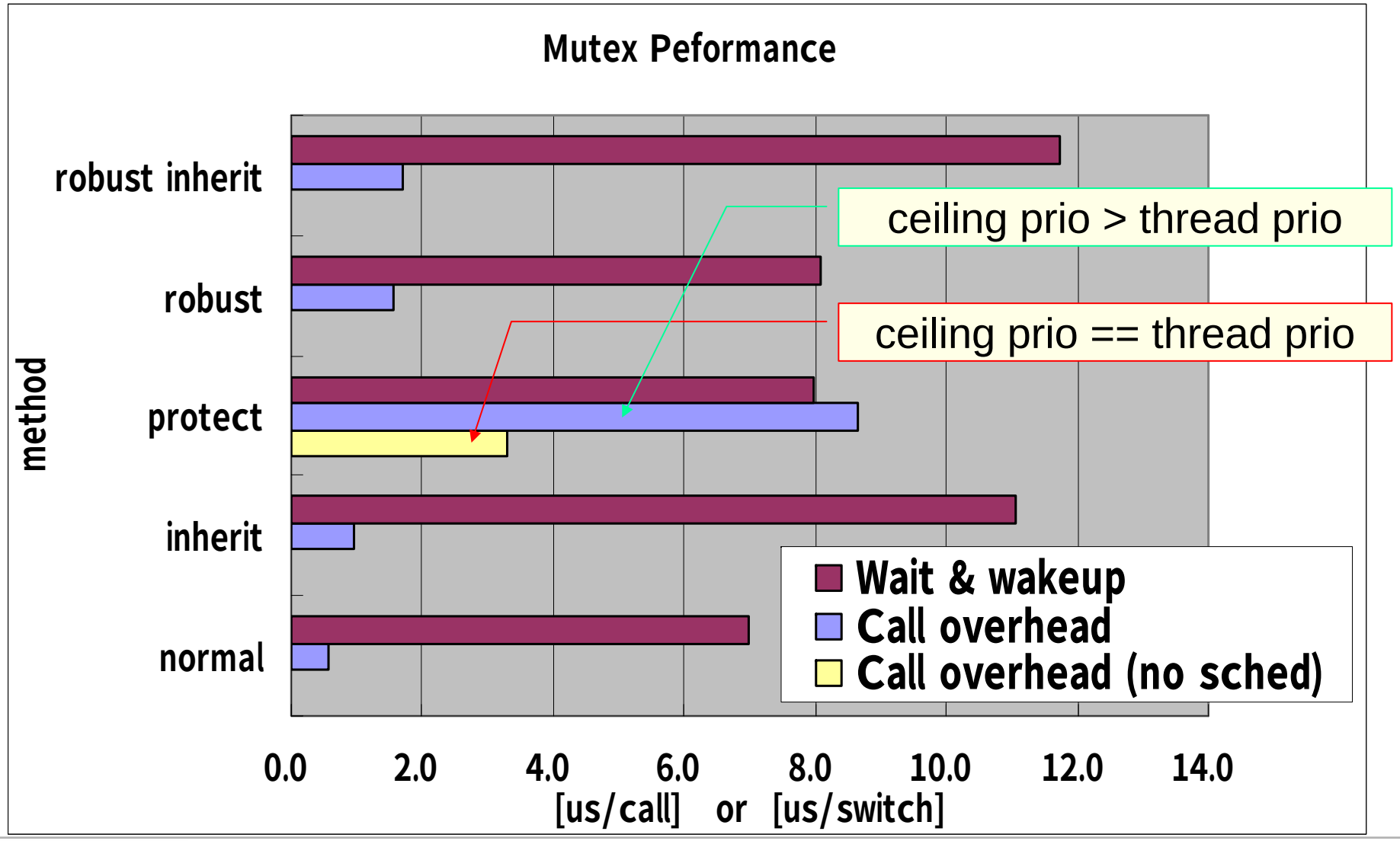


Wait & wakeup (slow path)



Mutex Performance

TX49/H4, MIPS 333MHz, D$ 32kB, I$ 32kB
kernel 2.6.20.12, gcc 4.1.2, glibc 2.5



Priority (Inheritance/Protect) Mutex まとめ

	Priority Inheritance Mutex	Priority Protect Mutex
非ロック競合時 **fast path**	*通常のmutexと比較して* **やや低下。** **1.5 倍程度。** △	*通常のmutexと比較して* **かなり低下。** **5〜15 倍増加。(*)** ×
	低下要因 – **glibc内のTID取得処理。**	*低下要因* – **内部ロック処理。** – **優先度変更処理。**
同期処理発生時 **slow path**	*通常のmutexと比較して* **やや低下。** **1.6 倍程度。** △	*通常のmutexと比較して* **同程度。** **1.2 倍程度。** ○
	低下要因 – **カーネル内のPI処理。**	*低下要因*
結論	**ロックが衝突する頻度が (衝突しない頻度場合よりも) 高いとは考えにくい (実装が悪い) ため、fast pathを重視すると、PPを使う理由は考えにくい。**	

TOSHIBA
Leading Innovation

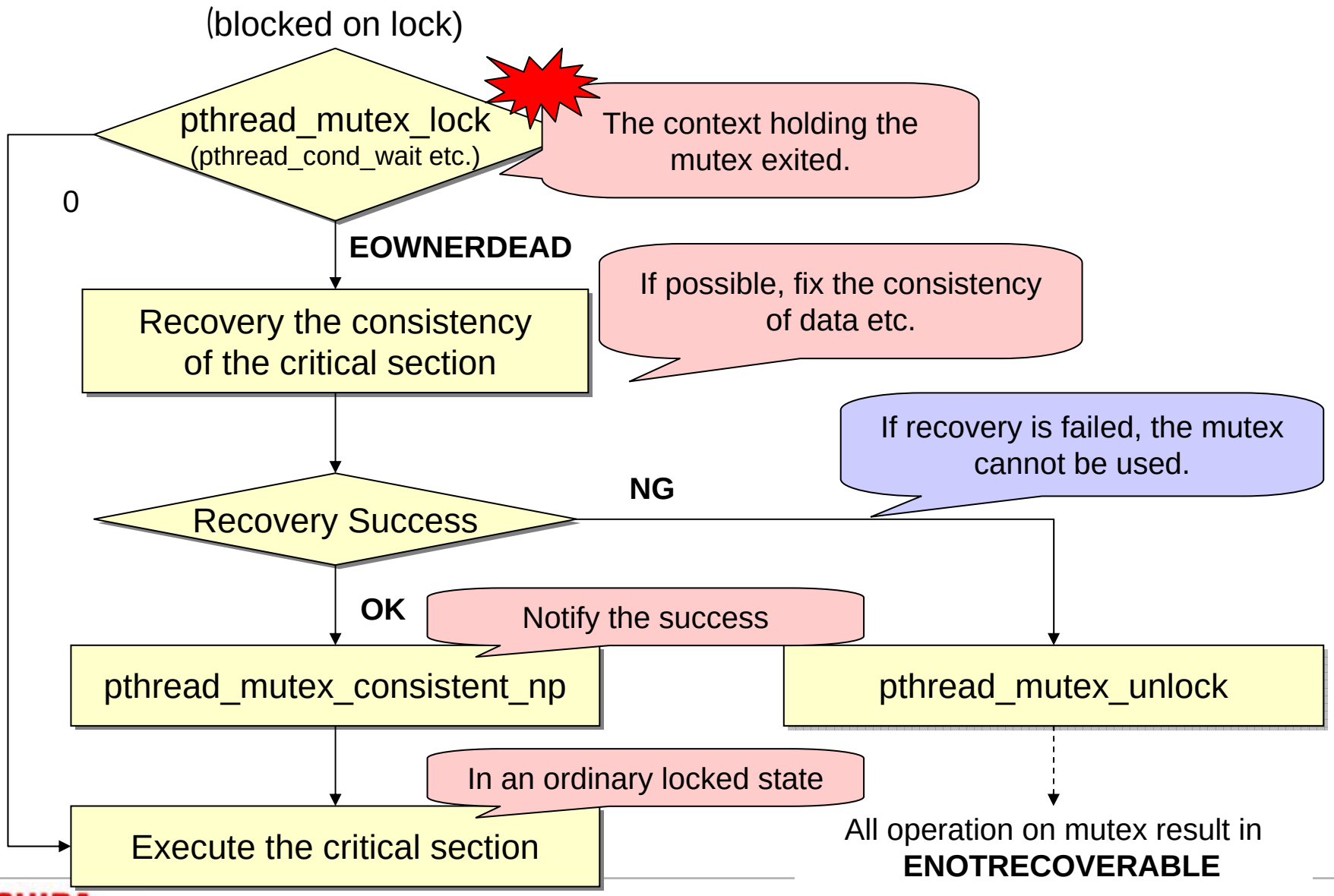
(*) シーリング優先度との関係に依存

Robust Mutex

Robust Mutex

	Robust Mutex
動作	スレッドがロックを持ったまま終了した場合、他のスレッドが その状態(owner dead) を検出できる。
特徴	プロセスとスレッドを組み合わせた信頼性のあるシステムの同期処理を構築できる。
サポート	kernel 2.6.17 ~ glibc 2.5 ~
注意	1. 複数スレッドがロック待ち中に owner dead が発生すると、2番目以降のスレッドの待ちが解除できなくなる。 "robust mutex does not work if owner died with multiple waiters" http://sources.redhat.com/bugzilla #4512 (fixed in glibc 2.5.1, 2.6.1) 2. static link で main thread のみが動作するプログラムで、nptl/init.oがリンクされない条件下では、Robust mutex が動作しない。 (例えば、main関数で mutex lock / unlock だけを行うプログラム。) 3. POSIX仕様策定中。

Robust Mutex - process flow



Inter-Process Synchronization with Robust Mutex

- **ここでは**
  Robust Mutex の機能ではなく、用途に注目。

プロセス間の同期処理

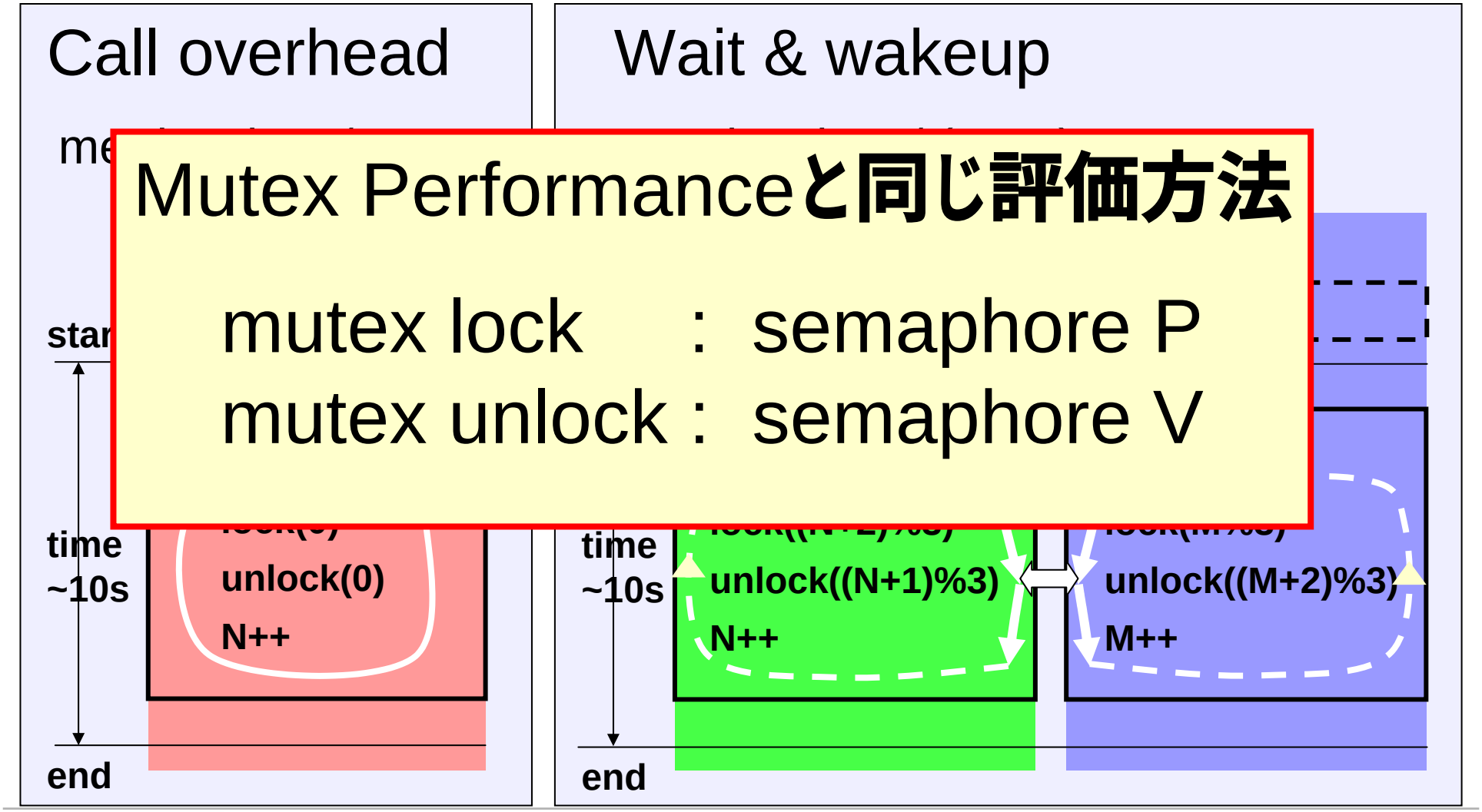
- **やりたい事**
  Robust Mutex をベースにしたプロセス間同期処理と 従来手法(IPC)を比較評価する。
  - **排他** (Mutual execution)
  - **メッセージ通信** (Message communication)

Inter-Process Mutual execution with Robust Mutex

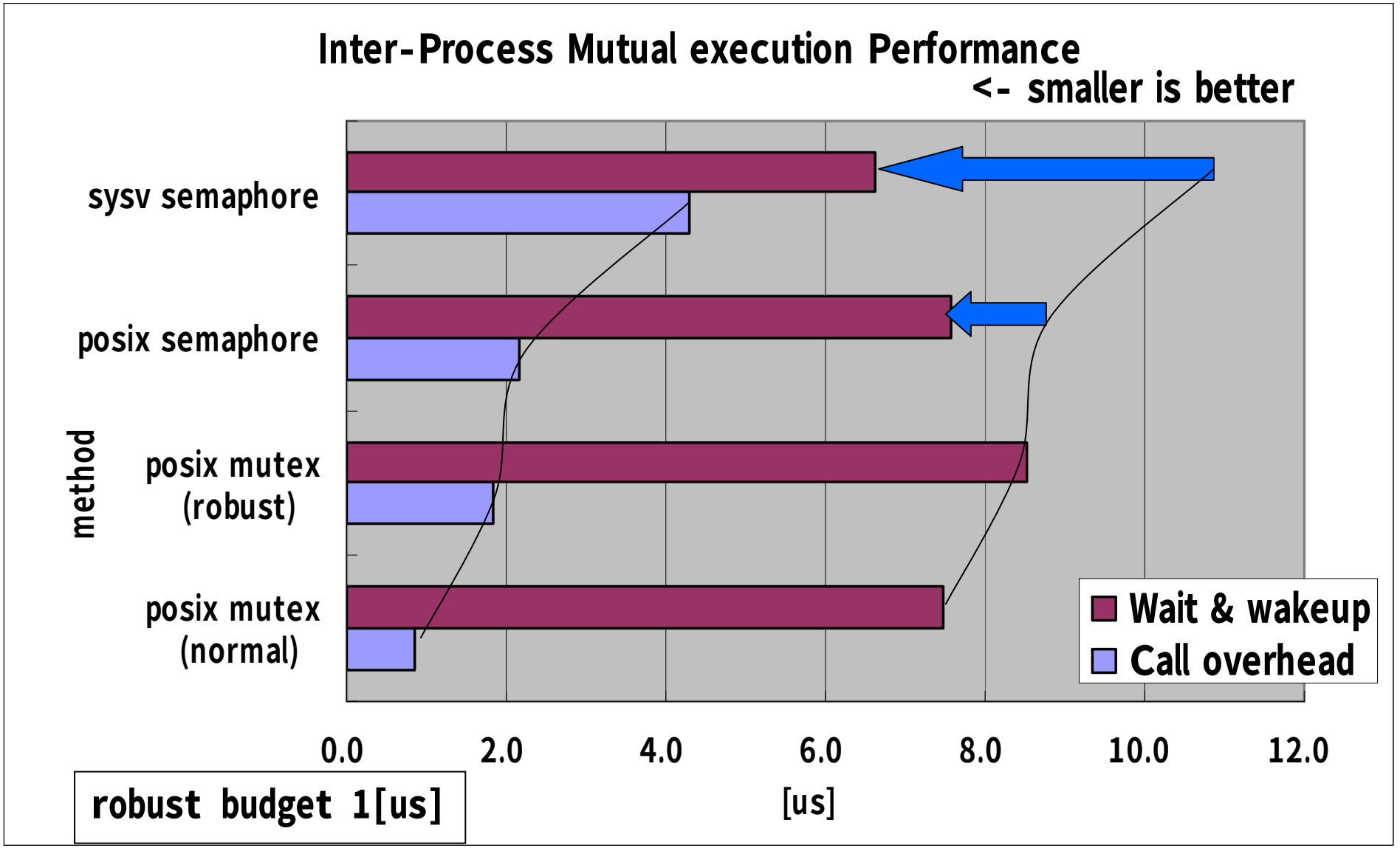
- **排他 (Mutual execution)**
  - **比較対象**
    - SysV IPC Semaphore (semX) with SEM_UNDO*
      ***プロセスが終了した場合にリソースが解放される機能を持つ。**
    - Robust mutex
      ※ 共有メモリをmmap**して同期変数を配置。**
  - **性能** (Performance)
    - **呼び出しオーバヘッド** (Call overhead)
      - **リソース競合なし、1プロセス、呼び出しオーバヘッド。**
    - **コンテキストスイッチ** (Wait & wakeup)
      - **リソース競合あり、2プロセス間のスイッチ。**

Inter-Process Mutual execution with Robust Mutex

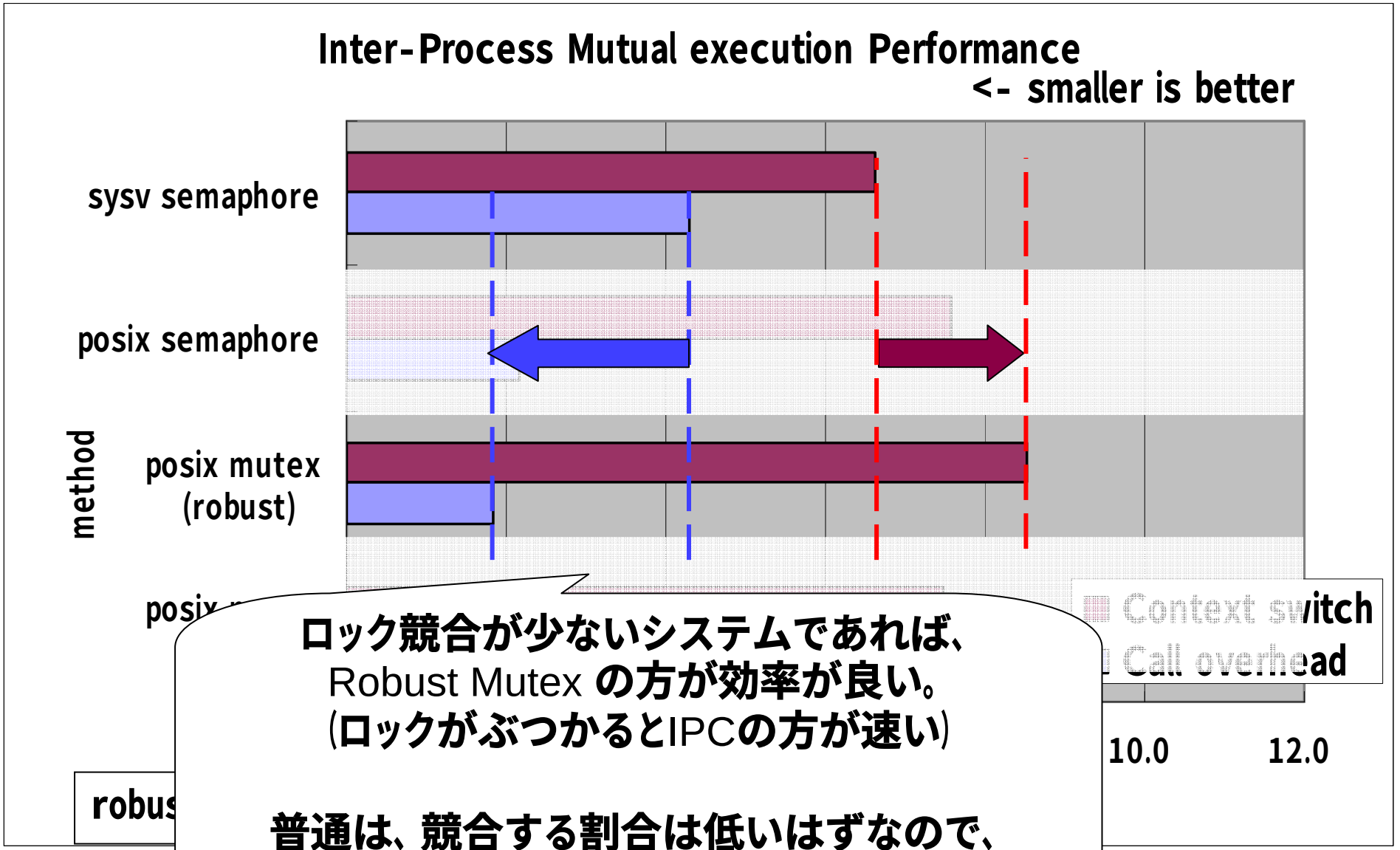
ex.) mutex (lock: P, unlock: V in semaphore)



Inter-Process Mutual execution with Robust Mutex



Inter-Process Mutual execution with Robust Mutex



ロック競合が少ないシステムであれば、
Robust Mutex **の方が効率が良い。**
(ロックがぶつかるとIPCの方が速い)

普通は、競合する割合は低いはずなので、
Robust mutex **が良い。**

Inter-Process Mutual execution with Robust Mutex

- **まとめ**

	Robust Mutex	SysV IPC Semaphore
非ロック競合時 fast path	**1.8[us]** ○	**4.3[us]** ×
	低下要因	*低下要因*
		– システムコールが必須
同期処理発生時 slow path	**8.5[us]** ×	**6.6[us]** ○
	低下要因	*低下要因*
	– 待ちリスト管理処理など (ユーザ空間側処理)	
性能以外	**同期リソースの回復処理を実行でき、自由度が高い。 優先度継承も併用可能。**	**同期リソース自動解放が可能 (SEM_UNDO)**
結論	**性能と機能の両面から、Robust Mutex が有用と言える。**	

Inter-Process Message Communication with Robust Mutex

- **メッセージ通信 (Message communication)**
  - **比較対象**

* 測定のため、メッセージバッファ合計上限を約2MBに変更。
MQ_BYTES_MAX 2101248 @ include/linux/mqueue.h

    - Posix Message Queue* (mq_X)

      システムコール実装、N:N**通信、メッセージ優先度。**

    - Message Queue with Robust Mutex + Sync Method
      - mu/cv ... Robust Mutex & **条件変数方式**
      - mu/sm ... Robust Mutex & **セマフォ方式**

      ※ 共有メモリをmmap**して同期変数を配置。**

  - **性能** (Performance)
    - **コンテキストスイッチ** (Wait & wakeup + msg copy)
      - **2プロセス間の2つのキューを使用した、双方向メッセージ送信。**

Inter-Process Message Communication with Robust Mutex

- **Message Queue with Robust Mutex + Sync Method**

mu/cv: **条件変数方式**	mu/sm：**セマフォ方式**
mq_send(,char *msg, size_t msg_size,)	

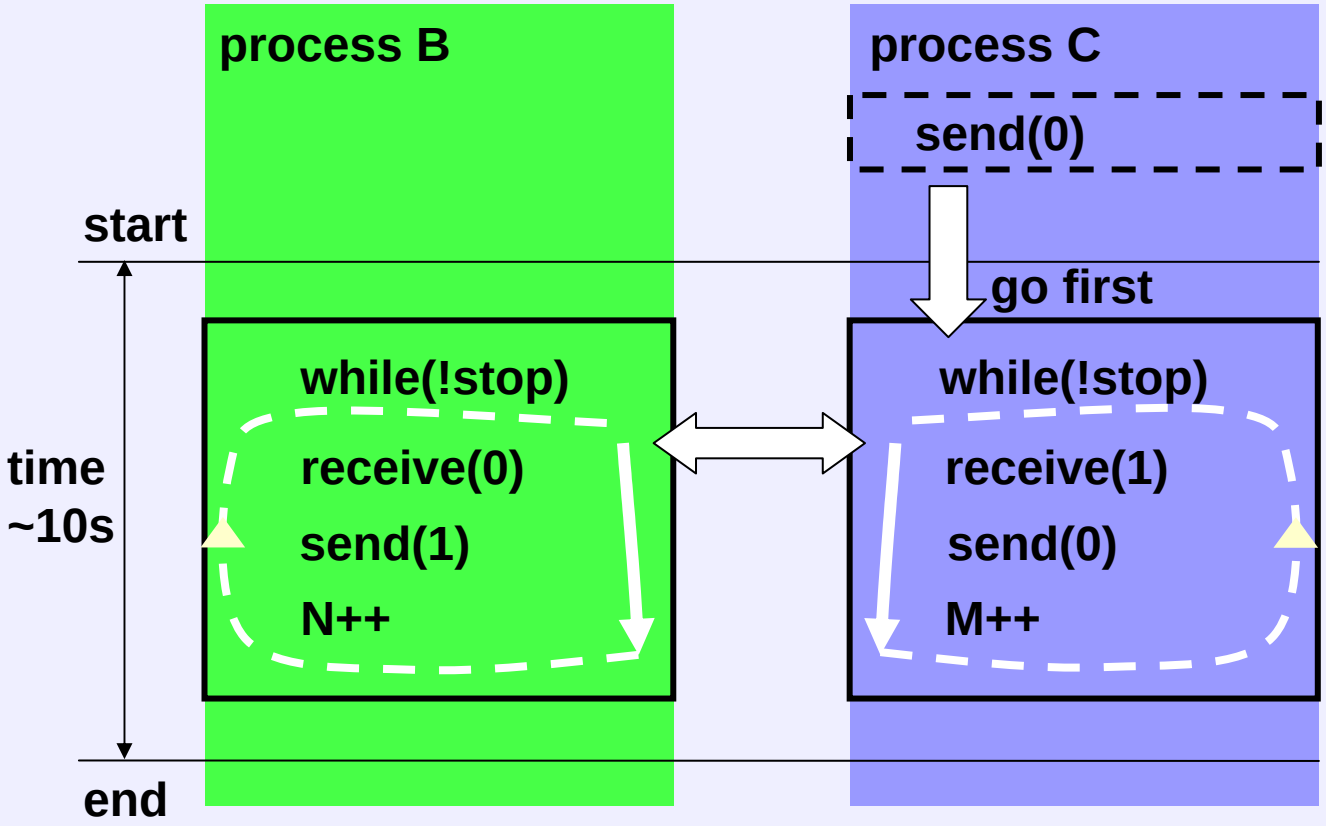
pthread_mutex_lock();
memcpy(shared memory, msg, msg_size);
count++;
pthread_cond_broadcast();
pthread_mutex_unlock();
	pthread_mutex_lock();
	memcpy(shared memory, msg, msg_size);
	count++;
	
	pthread_mutex_unlock();
	sem_post();

mq_receive(, char *msg, size_t msg_size,)	

pthread_mutex_lock();
while(!count)
pthread_cond_wait();
memcpy(msg, shared memory, msg_size);
count--;
pthread_mutex_unlock
	sem_wait();
	pthread_mutex_lock();
	
	memcpy(msg, shared memory, msg_size);
	count--;
	pthread_mutex_unlock

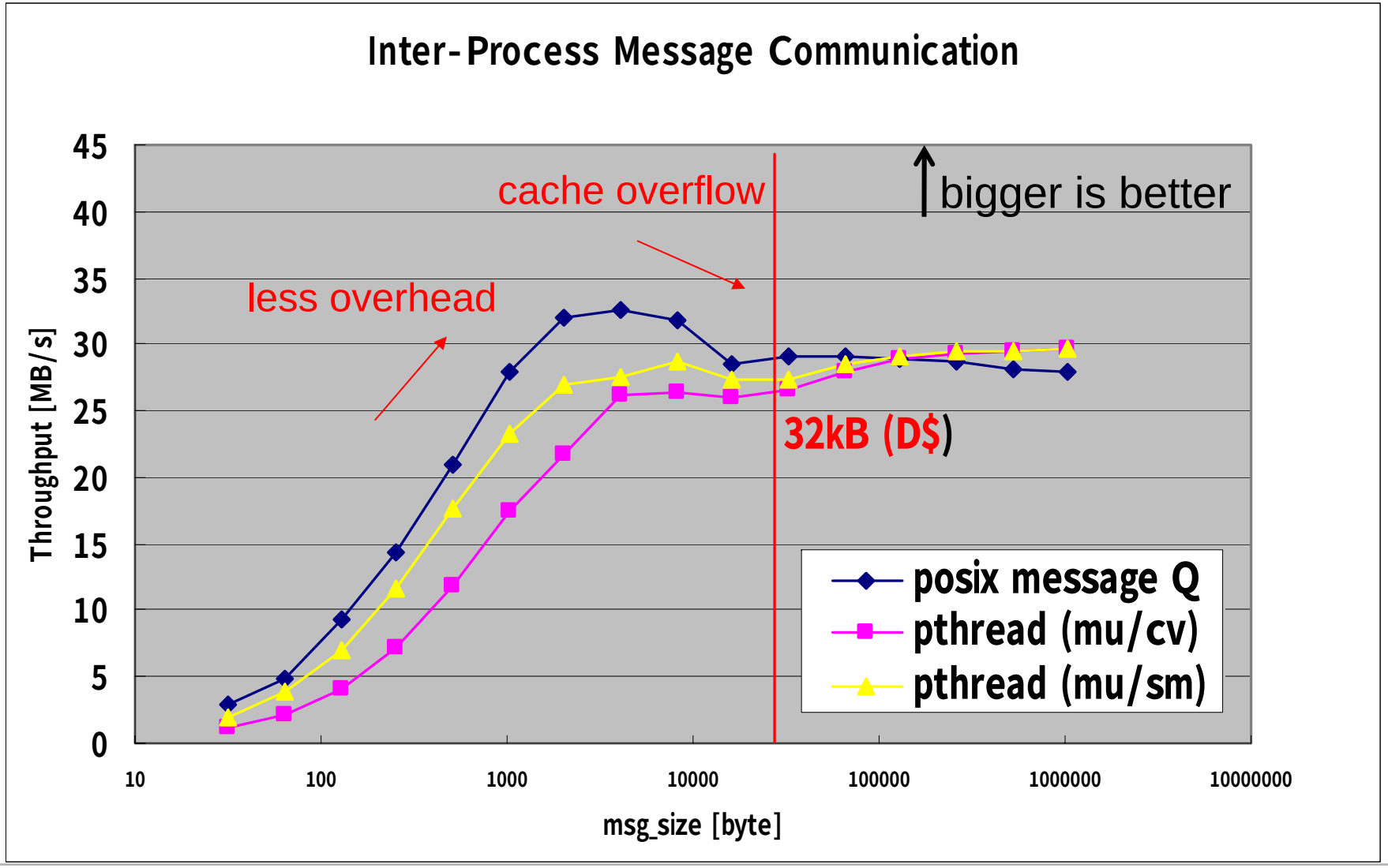
Message transfer throughput

metric: msg_size * time / (N+M)



Inter-Process Message Communication with Robust Mutex

- **性能結果**



Inter-Process Message Communication with Robust Mutex

- **まとめ**

	Robust Mutex		POSIX message queue
	条件変数方式 mu/cv	セマフォ方式 mu/sm	
スループット	×	△	○
性能以外	仕様をカスタマイズ可能		アクセス制限などセキュリティ的に有利。
結論	IPCが一番良い結果。 特定仕様(1:1通信)の通信処理などでは、スレッドライブラリ機能による実装も有用になるシーンはあると考えられる。 Robust Mutex + futex方式 を作れば、POSIX message queue と同程度までいけるかも！？		

まとめ

- **Priority (inheritance/protect) Mutex**
  - **基本概念から実装･性能まで紹介。**
  - **性能的には、PPは使用しなくても良さそう。**

- **Robust Mutex**
  - **用途としてのプロセス間同期処理への応用を評価。**
    (今回の評価環境では)
    - **排他なら**Robust Mutex。
    - **メッセージ通信なら**IPC。


TOSHIBA
Leading Innovation