Linux のスケジューリング調査と超低優先度プロセスの実装

中谷 翔

1/25/2011

動機

動機

- 「テスト勉強なしとかマジ俺得 www」

現実

現実

- テスト勉強 5 周はできたんじゃ …

現実

- テスト勉強 5 周はできたんじゃ …
- これで優がこないのはねぇ …

この資料の配布先

- **自鯖**
  http://www.laysakura.net/oskadai.pdf
  - **一斉アクセスとかやめてください><**
  - **サイバー攻撃しないでください><**
- eeic **のう** p **ろだにもあります**
  - **ろだ** -> OS -> nakatani_presen.pdf

やったこと

- Linux のメインのスケジューリング機構 CFS (Completely Fair Scheduler) の調査
- CFS のハック

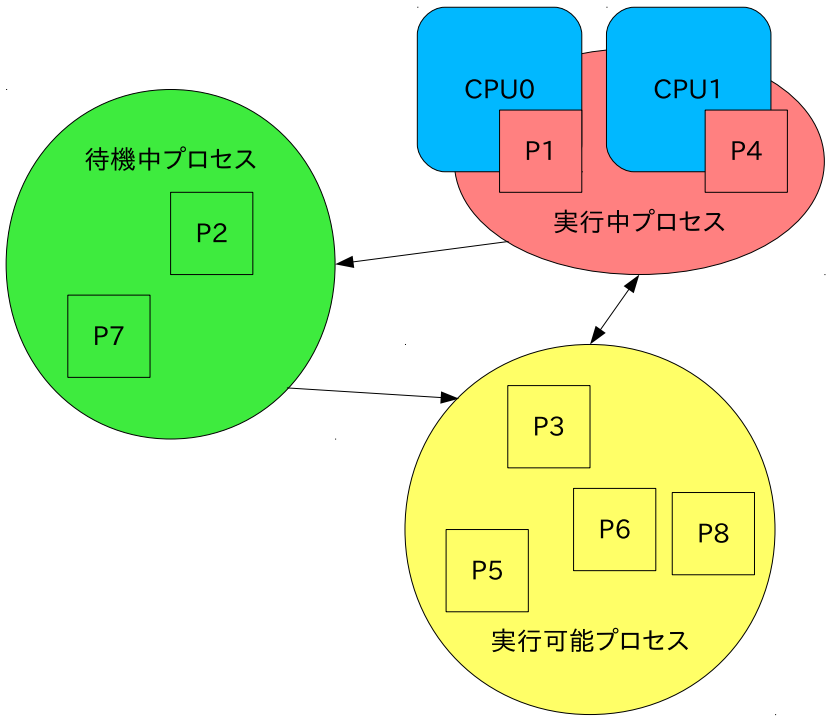
開発環境

- **カーネル**
  - 2.6.36.2
    先日 2.6.37 **が出るまでは最新の** stable **だった**
- **ディストロ**
  - Debian GNU/Linux 5.0
- **仮想化**
  - kvm & qemu 0.12.5
    - CPU1 **コア**

諸注意

- 分かりづらい話が多いかも
  - 質問は随時受け付けます
  - あまりに流れをぶった切る質問は最後にお願いします
- 喋りだすと止まらない方です///
  - 適宜「巻いて」ジェスチャーをお願いします
- 最後の方のページでカーネルハック入門を共有したので，是非今後の参考にどうぞ

スケジューリングの基礎



- 1 つの CPU は 1 つのプロセスしか同時に実行できない
  - 図では P1，P4 のみが実行中
- 「実行可能」から「実行中」に移行するプロセスを選択し，移行のタイミングも決めるのがスケジューラの役割

Linux におけるスケジューリング

- 基本的には前ページの図の通り
- スケジューリングの契機:
  - 実行中プロセスが自ら実行権を手放す
  - 実行中プロセスに割り当てられた実行時間を使い果たしたとき
  - プリエンプション
    - 新たなプロセスが立ち上がったとき，そのプロセスの方が実行中プロセスより優先度が高いときに発生
- 「実行可能」から「実行中」に移行するプロセスを選択するアルゴリズム
  - -> CFS (Completely Fair Scheduler)
    - CFS について見る前に,「プロセスの切り替え (context switch)」の実装を見てみる

プロセス切り替えの実装: switch_to マクロ

- 今回の発表的には余談みたいなものですが，面白いので．

arch/x86/include/asm/system.h: switch_to マクロ

```
#define switch_to(prev, next, last)                     \
do \{                                                   \
    unsigned long ebx, ecx, edx, esi, edi;              \
    asm volatile("pushfl\n\t"       /* save    flags */ \
        "pushl %%ebp\n\t"       /* save    EBP   */ \
        "movl %%esp,%[prev_sp]\n\t" /* save    ESP   */ \
        "movl %[next_sp],%%esp\n\t" /* restore ESP   */ \
        "movl $1f,%[prev_ip]\n\t"   /* save    EIP   */ \
        "pushl %[next_ip]\n\t"  /* restore EIP   */ \
        ...
        "memory");                      \
\} while (0)
```

- CPU のレジスタには，実行中プロセス prev の情報 (Instruction Pointer, Stack Pointer, その他汎用レジスタ) が乗っている．それを一旦メモリ上に退避させて，レジスタを next 用に空けるのが主な仕事．

CFS(Completely Fair Scheduler) の周辺知識

- kernel-2.6.23 から現行のバージョンに至るまで使用している，スケジューラの実装のひとつ
- 本当に completely fair かは置いといて，評判がよいのでここまで続いてる
  - ベンチマークの結果は，他のスケジューラ実装と比べても良いらしい
- 全実装は，kernel/sched_fair.c にある
  - 3921 行でした
  - ただし，CFS 以外のスケジューラもあるので注意

CFS: 基本戦略 (1) - 各プロセスへの CPU 割り当て時間

- 全 RUNNABLE プロセスを一周スケジューリングさせる (とした場合の) 周期を一定にする
  - ただし，プロセスが多くなりすぎると，プロセスの切り替え回数が多くなりすぎる
    - プロセスが多すぎて，プロセス切り替えが多くなりすぎるときは，プロセス当たりの時間に最小値を
- 上記戦略に基づいて計算された時間を，各プロセスへ割り当てる
  - ただし，上記の「一周時間」に本当に全ての RUNNABLE なプロセスが CPU に乗るわけではない
    - CPU に乗るプロセスは，次のような戦略に基づいて決定している

CFS: 基本戦略 (2) - CPU に割り当てるプロセスの選択

- RUNNABLE なプロセスのうち，「得点」(vruntime) が最も低いプロセスを割り当てる
  - この vruntime こそが「優先度」の実体
- CPU に乗ったプロセスにはペナルティを，乗らないプロセスにはボーナスを与える
  - CPU に乗った時間分，vruntime をプラスする
  - WAIT した (sleep した) 時間分，vruntime をマイナスする

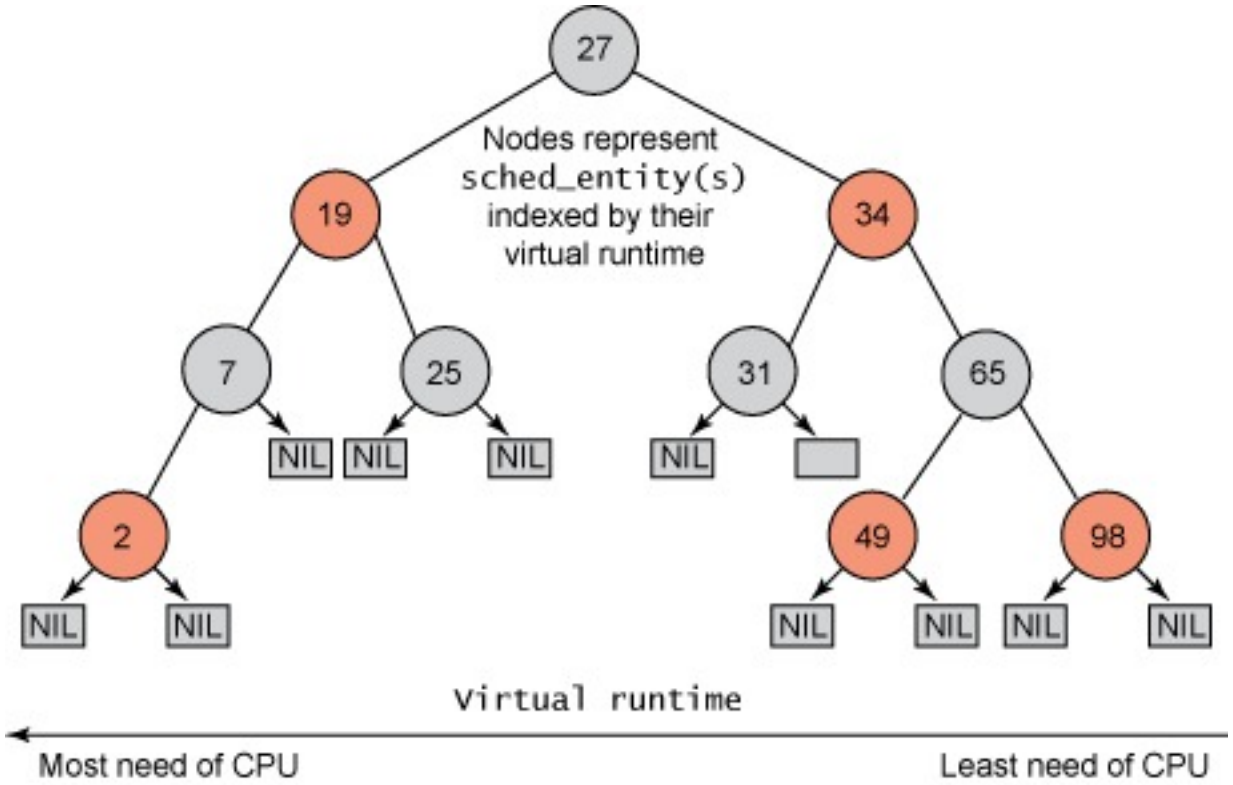
各プロセスへの CPU 割り当て時間の計算 (1)

- **全 RUNNABLE プロセスを一周スケジューリングさせる (とした場合の) 周期を一定にする**
- **周期の指標**: sched_latency = 5 msec * (1+log2(CPU **数**))
  - 5 msec **は** kernel/sched_fair.c **中のグローバル変数** sysctl_sched_latency
  - (1+log2(CPU **数**)) **は，** kernel/sched.c **中の関数** get_update_sysctl_factor()
- **周期**: period = 0.625 msec * RUNNABLE **プロセス数** (RUNNABLE **プロセスが多すぎるとき**) **または** period = sched_latency
  - **プロセスへ割り当てる時間の最小値，**0.625 msec **は，** kernel/sched_fair.c **の中のグローバル変数** sysctl_sched_min_granularity
  - RUNNABLE **プロセス数は，** kernel/sched.c **中** cf_rq->nr_running

各プロセスへの CPU 割り当て時間の計算 (2)

- **各プロセスの割り当て時間** = period * weight/total-weight
  - weight **は，各プロセスの重み．**nice **値から算出される．**
  - **「各プロセスの割り当て時間」を** time slice **という**

CPU に割り当てるプロセスの選択 (1)



- vruntime **の最も小さいプロセスから** CPU **に割り当てられる**
  - **プロセスは，**vruntime **の大きさをキーとして赤黒木に並べられる**

CPU に割り当てるプロセスの選択 (2)

- vruntime は，CPU に乗っている時間に対応
- 立ち上がったばかりのプロセスは，赤黒木中で最も小さい vruntime より少しだけ大きい値を割り当てられる
  - すぐに CPU に乗せてもらえる
  - 「少しだけ大きい値」は，自分の time slice から算出される

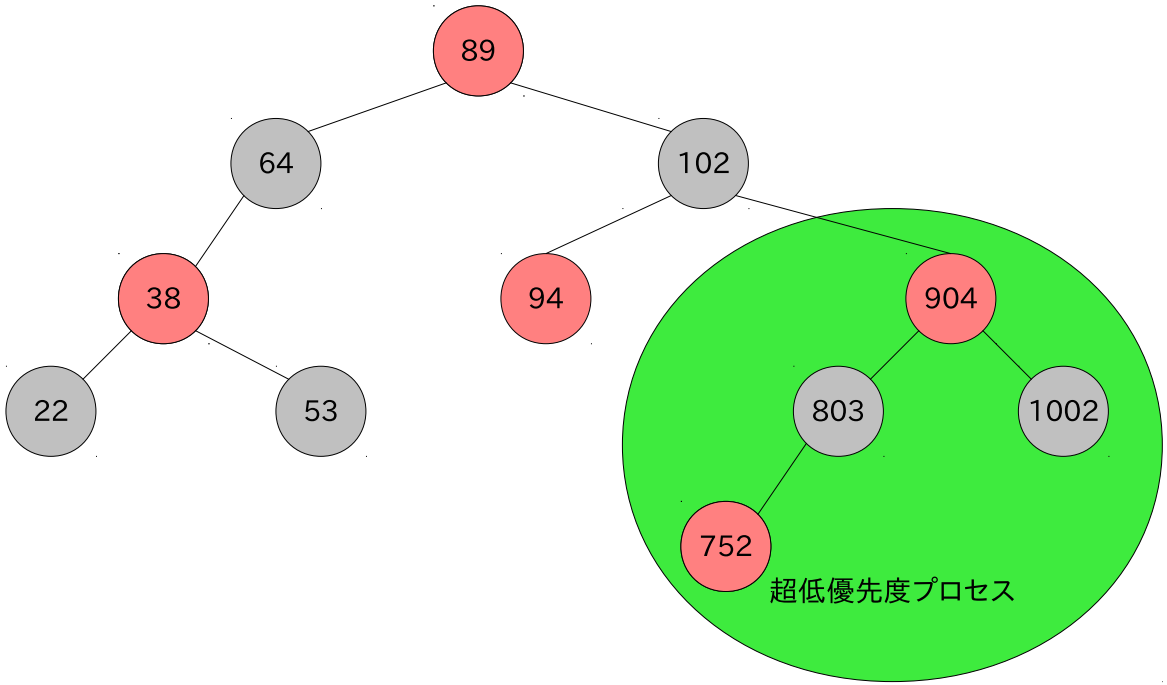
何をした?

- **超低優先度プロセスの実装**

超低優先度プロセスの意義

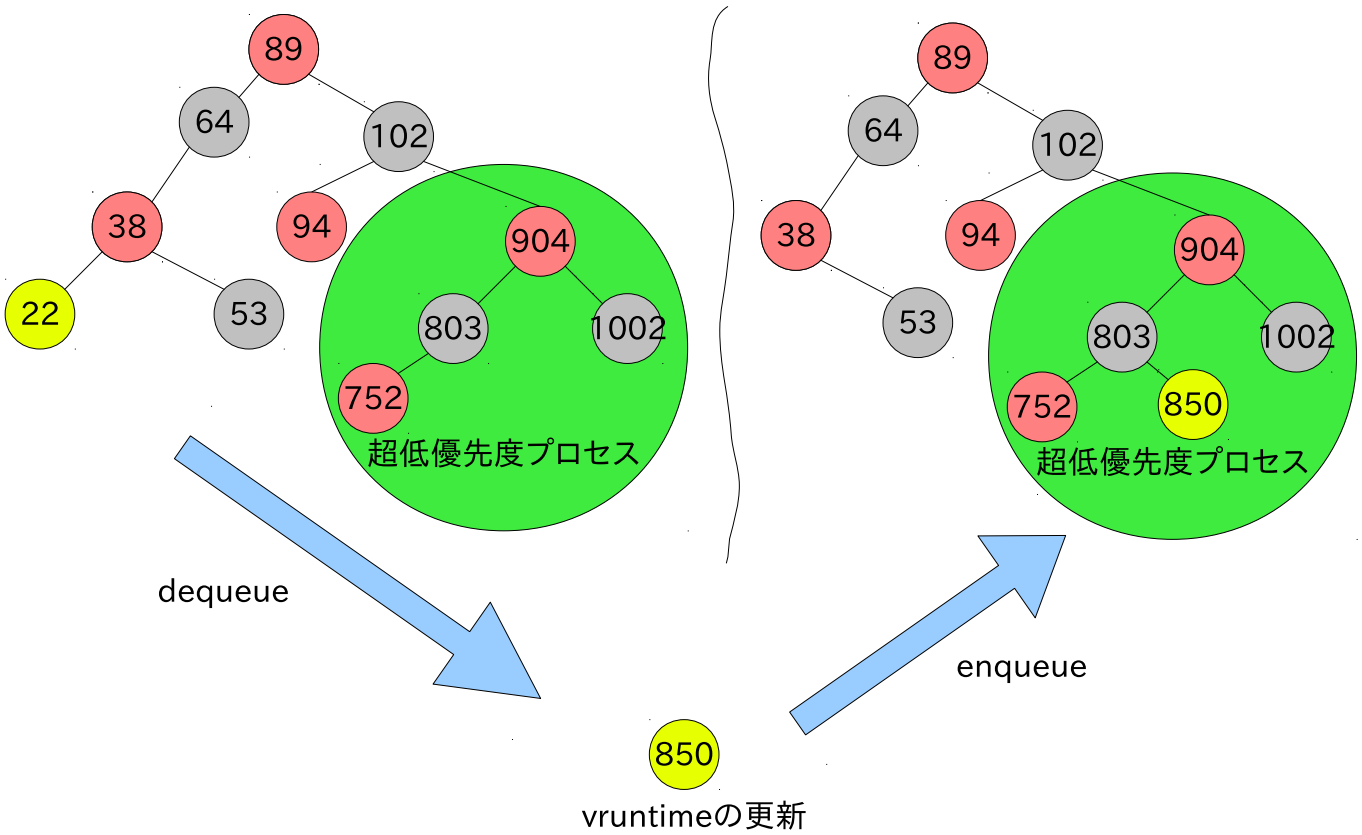
- 他のプロセスがいるときは絶対スケジューリングされない
  - マシンリソースを共有している場合，プロセスを立ちあげっぱなしにしても怒られない

実装方法



- 低優先度プロセスは，必ず通常のプロセスよりも vruntime が大きくなるようにする．
  - 実質上，赤黒木を二分する
- 結果，低優先度プロセスが実行中になるのは，通常のプロセスが赤黒木上にないときのみ

実装方法 (詳細 1)



- プロセスを，超低優先度モードに移行させるシステムコールを追加

実装方法 (詳細 2)

- システムコールを発行したプロセスを，赤黒木から dequeue
- vruntime を「かなり大きく」する
  - vruntime は 64 ビット unsigned．最上位ビットが 0 のものを通常のプロセスに，1 のものを超低優先度プロセスに当てる．
    - この実装でも，vruntime のオーバーフローには約 250 年かかる
- 赤黒木に enqueue
- 以後のスケジューリング時にも，低優先度プロセスの vruntime は「かなり大きく」保つ

結果のデモ (1)

結果のデモ (1)

- 未完成です
- バグが取れません

結果のデモ (1)

- 未完成です
- バグが取れません
- カーネルパニック起きたらデモを終了します (笑)

結果のデモ (2)

- **赤黒木を可視化しました**
  - cat /proc/sched_debug **で見れます**

結果のデモ (3)

- 低優先度プロセスは確かに通常のプロセス走ってる時に止まります

結果のデモ (4)

- 低優先度プロセスを走らせ続けると...

ハイパー言い訳タイム

- スケジューラのデバッグってほんと難しいよ?
  - スケジューラ関数はものすごい頻度で呼ばれる
- 落ちる条件を予測 (5 分)

  コード修正 (10 分) カーネルコンパイル (2 分) カーネルインストール (2 分) テスト (1 分) 落ちる (一瞬)
- 心が折れないわけがない

学んだこと

- カーネル空間のプログラミングも，やることはユーザーランドと同じ
  - コーディング然り，デバッグしかり
- CFS は俺に聞け (笑)
- Linux カーネルのコードの綺麗さ，汚さ
  - sched.c は，スケジューラを統括した綺麗なインターフェイス
  - sched_fair.c は，同じような処理が点在していて非常に読みづらい

おまけ

- Follow me on Twitter
- @laysakura
- **ご感想などこちらへ ^^**

質疑応答タイム

- 何でもどうぞ
  - カーネルの話
  - スケジューラの話
  - その他プログラミングの話
- お恥ずかしければ twitter にでも

終わり

- ありがとうございました.
  テスト勉強頑張ってください ^^

VM の導入

- Q. VM**って**?
  - A. **仮想マシン** (Virtual Machine)．KVM，QEMU，VMWare，VirtualBox **などが有名**
- Q. **何で必要なの**?
  - A. **カーネルに手を加えたことで，メインのマシンでブートできなくなると悲惨**

qemu-kvm + virt-manager の導入

- virt-manager **は，** qemu-kvm **を扱うための** GUI **ツール**
  - **ないと結構大変**
- host:# apt-get install qemu-kvm virt-manager
- host:# virt-manager
  - **ディストロのイメージを使って，新しく仮想マシンを作成**
  - virt-manager **上で，ディストロをインストール**

virt-manager による qemu-kvm の設定

- host 側から gdb を guest 側に繋いでデバッグできるようにする
  - host:# virsh edit <VM-name>

変更箇所

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
...
    <serial type='pty'>
      <target port='1'/>
    </serial>
...
  <qemu:commandline>
    <qemu:arg value='-s'/>
  </qemu:commandline>
```

guest と host の連携

- NFS でファイル共有
  - host:# apt-get install nfs-kernel-server
  - host:# emacs /etc/exports

追加

/dir/to/share 192.168.0.0/255.255.0.0(rw,no_subtree_check)

  - guest:# apt-get install nfs-client
  - guest:# mount -t nfs <HOST IP ADDRESS>:/dir/to/share /dir/to/mount
- host のターミナルから guest を操作
  - guest:# apt-get install telnetd
  - host:$ telnet <GUEST IP ADDRESS>
    - VM のコンソールで操作しても良いが，ログがすぐ流れてしまう

カーネルの入手方法

- http://www.kernel.org/ **から，**kernel-<version>.tar.(gz|bz2) **をダウンロード**
- **または** git clone

こんなに簡単，カーネルコンパイル

手順

- make *config : **マシンごとに設定**
- make : **コンパイル**
- make modules_install : **できたモジュールをインストール**
- make install : **出来たイメージ (カーネルの実体) をインストール**
- (mkinitramfs) : initramfs **を作成**
- update-grub : **ブートローダの再設定**
- reboot
- **祈る**
  - Kernel panic **の恐怖**

カーネルコンパイルの実際 (1) - 概観

- **仮想マシン (以下，guest) に合わせて設定する**
  - guest:$ make *config
- **コンパイルは高速なホスト側で**
  - host:$ make
- **インストールはもちろんゲストに**
  - guest:# make modules_install install
  - guest:# mkinitramfs
  - guest:# update-grub

カーネルコンパイルの実際 (2) - kernel config

- **マシンの環境を自動で検出し，設定** -> guest:$ make localmodconfig
  - kernel-2.6.32 **から対応**
  - **普通に** kernel panic **することも．過信すべからず．**
- **ブートが確認できてる**.config **がある** -> guest:$ make oldconfig
- **細かい設定**
  - guest:$ make menuconfig
  - **エディタで直接** .config **を編集**
    - **コンパイルを速めるために，いらないモジュールとかは** disable **にする**

カーネルコンパイルの実際 (3) - make

- 並列コンパイルオプションを使う
  - host:$ make -j4
    - コア数の 2 倍か 4 倍の数が良いと偉い人が言ってました
- キャッシュを利用する
  - コンパイルの出力を保存し再利用
    - ccache

カーネルコンパイルの実際 (4) - install

- **モジュールをインストール**
  - guest:# make modules_install
    - **モジュールオブジェクトファイル** (*.ko) **を** /usr/lib/modules/<version> **にコピーするだけ**
- **カーネルイメージをインストール**
  - guest:# make install
    - arch/<CPU **アーキテクチャ**>/boot/bzImage **を** /boot **にコピーするだけ**
- initramfs **を作成**
  - guest:# mkinitramfs
- **ブートローダを再設定**
  - guest:# update-grub
  - **ブートメニューで今コンパイルしたイメージを選択できるようにする**
- **以上の手順は，スクリプトにしておくのが便利**

カーネルインストール後

- **新しいカーネルの情報を確認**
  - guest:$ uname -a

カーネルデバッグ入門

- printk debug
  - **ユーザ空間での** printf debug **と同じ**
    - **カーネルのソース中に，** printk(KERN_DEBUG "hogehoge"); **を挿入**
    - **再構築したカーネルでは，**/var/log/syslog **にて出力が確認できる．**
- GDB **によるデバッグ**
  - host **から** guest **を** GDB **でデバッグできる**
  - .config **オプションで，**CONFIG_DEBUG_KERNEL, CONFIG_DEBUG_INFO **を有効に**
    - **カーネル再構築**
  - guest **を立ち上げる**
  - host:$ gdb linux-<version>/vmlinux
    - (gdb) target remote localhost:1234
    - **後はいつも通り** GDB **を操作**

システムコールの追加方法: 注意

- カーネルのバージョンによって追加方法が異なる
  - ネットにも情報が少ない
- 既存のシステムコールの定義・宣言を参考に
  - find -name '*.[ch]' |xargs grep -n vfork とかで，慎重に追加箇所を特定

システムコールの追加方法: kernel-2.6.36.2 の場合 (1)

- arch/x86/include/asm/unistd_32.h **の編集**

追加箇所

```
#define __NR_prlimit64          340
#define __NR_lowprio            341

#ifdef __KERNEL__

#define NR_syscalls 342
```

システムコールの追加方法: kernel-2.6.36.2 の場合 (2)

- include/asm-generic/unistd.h **の編集**

追加箇所

```
...
#define __NR_unshare 97
__SYSCALL(__NR_unshare, sys_unshare)
#define __NR_lowprio 341
 __SYSCALL(__NR_lowprio, sys_lowprio)
...
#undef __NR_syscalls
#define __NR_syscalls 265
...
```

システムコールの追加方法: kernel-2.6.36.2 の場合 (3)

- arch/x86/kernel/syscall_table_32.S **の編集**

追加箇所

```
.long sys_prlimit64     /* 340 */
.long sys_lowprio
```

- arch/x86/kernel/process.c **の編集**
  - **もちろん，追加するファイルは自分のシステムコールの動作によって決める**
  - **システムコールの関数名は，**sys_hoge **であることに注意．**
    (例) sys_fork

追加箇所

```
void sys_lowprio(int switch_to_low)
{
    if (switch_to_low)
        current->se.islowprio = 1;
}
```