logbook of NetBSD/aarch64

AsiaBSDCon2018 - NetBSD Bof

ryo@nerv.org

とりあえずNetBSD/aarch64の現状

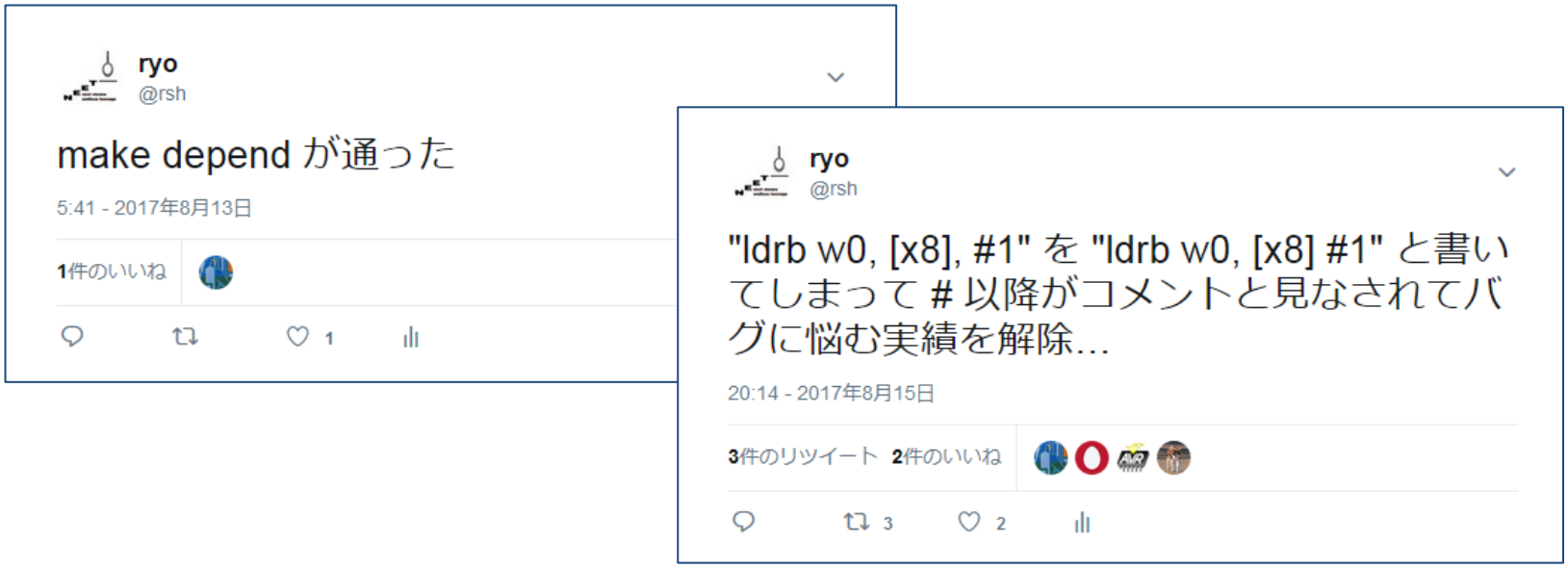
- current status
  - multiuser works
  - fdt support (refer evbarm/conf/GENERIC64)
  - kernel はそこそこ安定
  - commitに向けて現在cleanup中
- known bugs
  - __thread attributed variables fail to work (ld_elf.so issue?)
  - C++ stack unwinding (libunwind?)
    - C++で書かれているATFが動かない…

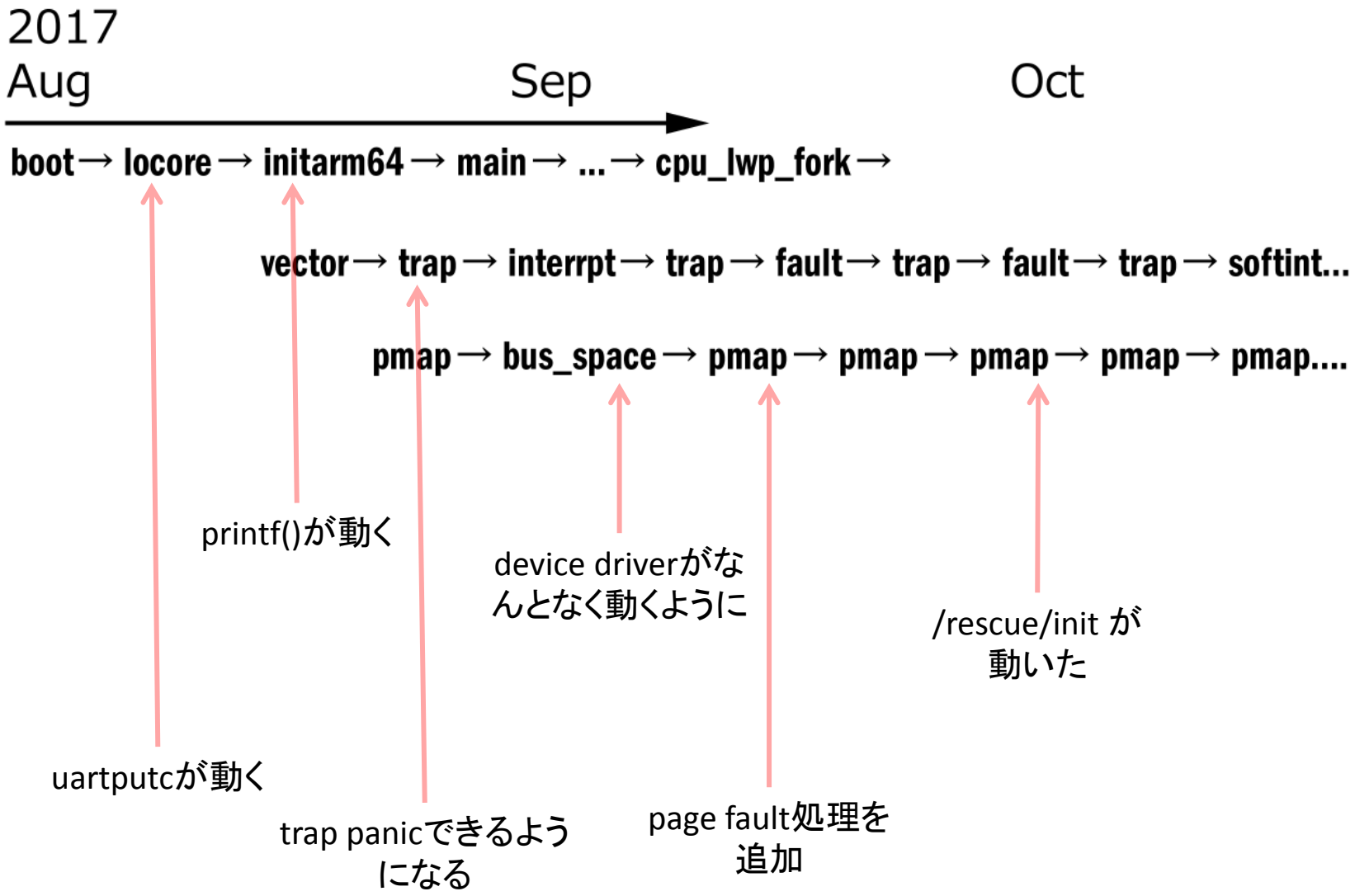
とりあえずNetBSD/aarch64の現状

- not yet
  - SMP
  - kernel text/rodata segments are mapped writable. it should be mapped as RX when no options DDB
  - ucas(9) and ucas_ras_check() is not implemented yet
  - crash(8), savecore(8) and libkvm (cpu_kcore_hdr_t)
  - pmap should be work even if PID_MAX >= 65536
  - TLB ASID in pmap should be randomized
  - userland debugger
  - COMPAT_NETBSD32
  - and so on…

実装について

- やったこと
  - matt@ による evbarm64/conf/A64EMUL が既にあり、build releaseはできていた→build関連やuserlandはok?
  - trapやpmap等、各種重要な関数が空だった
  - （この時点では気づいてなかったが）他にも所々バグが…
  - まぁなんとかなるだろうと nisimura@ さんと作り始める




2017
Aug
Sep
Oct
boot → locore → initarm64 → main → ... → cpu_lwp_fork →
vector → trap → interrpt → trap → fault → trap → fault → trap → softint...
pmap → bus_space → pmap → pmap → pmap → pmap → pmap....
printf()が動く
device driverがなんとなく動くように
/rescue/init が動いた
uartputcが動く
trap panicできるようになる
page fault処理を追加

locoreでのMMU設定

- FreeBSDのlocoreを参考にしつつ、まずはPA=VAでMMUをenableにする
- MMU onの状態に遷移して仮想アドレスにジャンプさせる

- MMU table (memory map)
  - MMUのL0～L2テーブルを作って、MMU enable

VA=PA	0x00000000_00000000～0x00000000_ffffffff =0x00000000_00000000～0x00000000_ffffffff	4G
KVA=PA	0xffffffc0_00000000～0xffffffc0_00?????? =0x00000000_00000000～0x00000000_00??????	kernel size
devmap	0xffffffff_f0000000～0xffffffff_ffe00000 =device physical address	
KSEG※	0xffff0000_00000000～0xffff007f_ffffffff =0x00000000_00000000～0x0000007f_ffffffff	512Gb

※FreeBSDではDMAP。MIPSのKSEG0と同じ使い方

locoreでこのテーブルを作り、VA=PAのテーブル以外は、ずっと使い続ける
（armはarm32_kvminitで作り直す）


user space
user program
or
VA=PA kernel at boot
0x00000000_00000000 VM_MIN_ADDRESS
256Tbyte
user space end
0x0000ffff_fffff000 VM_MAXUSER_ADDRESS
unused
KSEG
0xffff0000_00000000 AARCH64_KSEG_START
KSEG (direct map)
512Gbyte
0xffff0080_00000000 AARCH64_KSEG_END
unused
kernel VM
0xffffffc000000000 VM_MIN_KERNEL_ADDRESS
kernel text+data+bss
kernel vmspace
(malloc, pool, …)
512Gbyte
0xffffffff0000000 KERNEL_IO_VBASE
devmap
(early bus_space_map)
0xfffffffffe00000 VM_MAX_KERNEL_ADDRESS

ryo
@rsh

やっとinitarm()まで来た。しかし先は長い…

4:04 - 2017年8月20日

2件のリツイート **2**件のいいね 

1 2 2

別のツイートを追加

hashimoto kenichi @h_kenken · 2017年8月20日
返信先: @rshさん
期待

必要最低限のpmapを作る

kernelのmain()が動きだすために最低限必要なpmap関数

pmap_bootstrap	初期化処理その1(物理メモリサイズの設定等) main()を呼ぶ前に呼ぶ
pmap_steal_memory	物理メモリ範囲から必要なサイズを取る(盗む)
pmap_growkernel	仮想アドレス領域を増やす
pmap_extract	仮想アドレス→物理アドレス変換
pmap_kenter_pa	仮想アドレス→物理アドレスの割り当て(WIRED)
pmap_kremove	仮想アドレス→物理アドレスの割り当ての解除(WIRED)
pmap_init	初期化処理その2(この時点でpool cacheが動く) uvm_init()から呼ばれる

このへんまで作るとuvmの初期化が動くようになり、カーネルでkmem_alloc等のメモリ確保(kmem(9)、pool_cache(9)、malloc(9)等)ができるようになる

aarch64のMMU

- アドレス空間は48bit (256Tbyte)
- 4レベルハードウェアページテーブル
- ページ単位には4/16/64kbyteのページを選べるが、混在はできない（level1、level2で終端はできる）
- NetBSD/aarch64では4kbyte/pageで使用
  - level0 512Gbyte/table
  - level1 1Gbyte/block or table
  - level2 2Mbyte/block or table
  - level3 4kbyte/page

aarch64のMMU

- TTBR0/TTBR1レジスタ(**T**ranslation **T**able **B**ase **R**egister) でアドレスにより二つのMMUテーブルを使い分ける
- 48bit目でTTBR0(userland)/TTBR1(kernel)を使い分けるよう設定
  - 0x0001_0000_0000_0000～0xffff_ffff_ffff_ffff
    → TTBR1(kernel mode)
  - 0x0000_0000_0000_0000～0x0000_ffff_ffff_ffff
    → TTBR0(user mode)

- TTBRに、テーブルの物理アドレスを設定する
  - TTBR0 = user mode level0 table Physical Address
  - TTBR1 = kernel mode level0 table PA
- ページテーブルは8byte/entryで構成され、4kbyte/pageの場合は512 entry/table
- ページエントリには次のテーブルの物理アドレスと各種属性（read/write/exec/cache等）を格納

TTBR0/TTBR1

level0 page table page	
[0]	level1 table PA
[1]	level1 table PA
:	level1 table PA
[511]	level1 table PA

level1 page table page	
[0]	level2 table PA
[1]	level2 table PA
:	level2 table PA
[511]	level2 table PA

level2 page table page	
[0]	level3 table PA
[1]	level3 table PA
:	level3 table PA
[511]	level3 table PA

level3 page table entry	
[0]	PA + attr
[1]	PA + attr
:	PA + attr
[511]	PA + attr

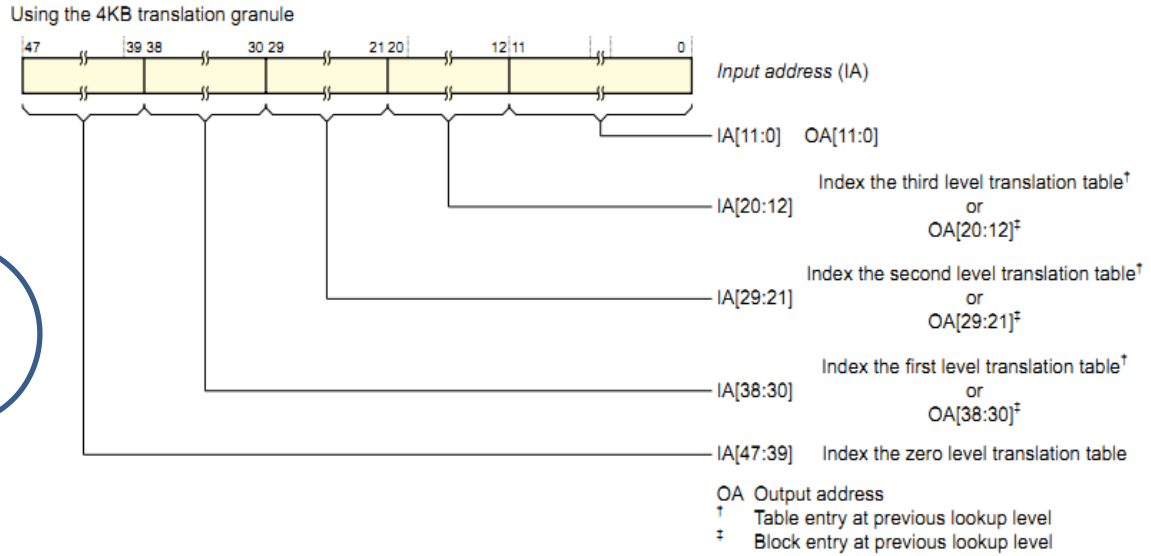


Figure D5-3 How the IA is resolved when using the 4KB translation granule

Figure D5-4 shows how a 48-bit IA is resolved when using the 16KB translation granule.

物理アドレス =
L0[index0]->L1[index1]→L2[index2]→L3[index3]

MMU tableとKSEG(DMAP)

- 仮想アドレスをを設定するには、MMU tableの適切な位置に書き込まなければいけない
- しかしMMUテーブルは**物理アドレス**で構成されている。物理アドレスしかわからない領域をkernelから仮想アドレスでアクセスするにはどうすれば…？

- 色々な方法があるが、netbsd/aarch64ではKSEG mappingを使っている
  - KSEG = MIPSのKSEG0セグメントと同じ機能
    物理アドレス範囲をそのまま仮想アドレスにマップした領域
  - FreeBSDではDMAPという名前を使っている
  - 物理アドレス→仮想アドレスをリニアにマップしている領域

[aarch64/include/vmparam.h]

```
#define AARCH64_KSEG_MASK       ((vaddr_t) 0xffff000000000000L)
#define AARCH64_KSEG_START      AARCH64_KSEG_MASK
#define AARCH64_PA_TO_KVA(pa)   ((vaddr_t) ((pa) | AARCH64_KSEG_START))
```

これにより、

```
va = AARCH64_PA_TO_KVA(pa);
```

とすれば、物理アドレス pa を仮想アドレス va でアクセスすることができる

MMU tableとKSEG(DMAP)

- 問題点
  - KSEGマッピングはPA→VAのアクセスが簡単にできて便利だが、全物理アドレスをread/writeでマッピングしているため、kernel内でのW^Xの前提が崩れる
    → セキュリティ的に問題がある？
  - OpenBSDでは使われていない。OpenBSDでは2つの連続するページを確保し、最初のページにMMUテーブル本体、2つ目のページに、内容をVAで写像した同様のテーブルを作って管理している。
  - NetBSDでもこれをやろうとしたが、pmap初期化時に連続する2ページをallocすることがうなくできなくて諦めた。
    → 頑張ればできると思われる。

exception and interrupt

- aarch64基礎知識

EL3	secure monitor
EL2	hypervisor
EL1	kernel mode
EL0	user mode

netbsdで面倒見るのはここから下

- EL0(user mode)やEL1(kernel mode)でsyscallやfault、その他変なことが起こるとEL1 synchronous exceptionが発生する

  - EL1は
    - spをEL0と共有する「EL1T」
    - spをEL0と共有しない「EL1H」

    の2つのモードがある。netbsdで使うのはEL1Hのみ
  - EL0は「32bit」と「64bit」の2つのモードがある
    (通常64bitのみ。COMPAT_NETBSD32を実装したら32bitもアリ)

  計4つのモードがあり、それぞれがsync、irq、fiq、errorの4つのexceptionを持っている

EL1T	sync	未使用(netbsdでは起きない)
	irq	未使用(netbsdでは起きない)
	fiq	未使用(netbsdでは起きない)
	error	未使用(netbsdでは起きない)
EL1H	sync	kernel modeでのfault等
	irq	kernel modeでの割り込み
	fiq	未使用(kernel modeでの高速割り込み)
	error	未使用
EL0_64bit	sync	user modeでのfault、syscall等
	irq	user modeでの割り込み
	fiq	未使用(user modeでの高速割り込み)
	error	未使用
EL0_32bit	sync	32bit user modeでのfault、syscall等(COMPAT_NETBSD32)
	irq	32bit user modeでの割り込み
	fiq	未使用(32bit user modeでの高速割り込み)
	error	未使用

interrupt(irq)はkernel modeでもuser modeでも同じ処理なので、必要な処理は

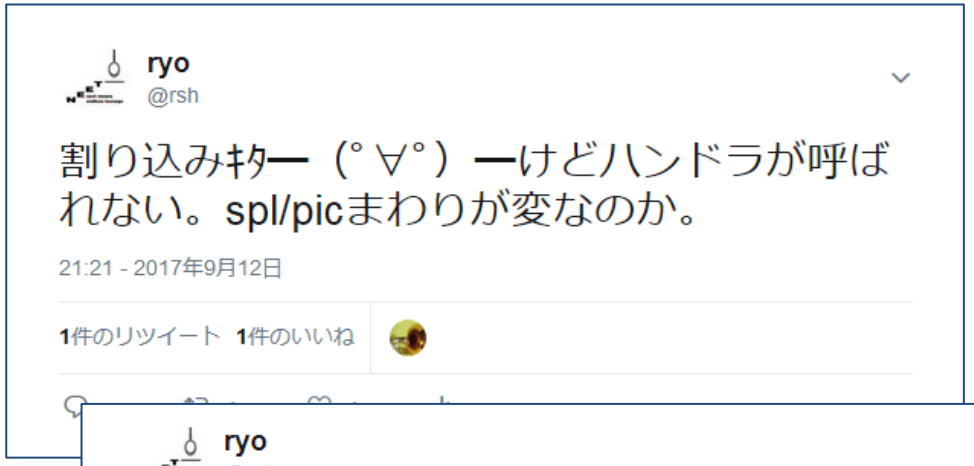
EL1H sync

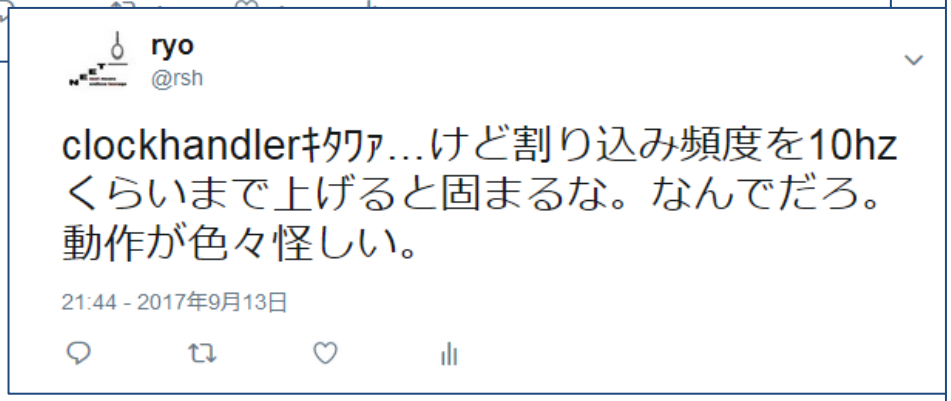
EL0_64bit sync

irq（処理は共通）

3つを実装すればよい

pmap、bus_space、irq exception → interrupt まわりまでできたら、最低限のデバイスドライバと割り込みが動くようになる





implementation of aarch64/pmap.c

pmap_create	プロセス毎のpmap instance作成
pmap_destroy	プロセス毎のpmap instance削除
pmap_enter	仮想アドレス→物理アドレスの割り当て
pmap_remove	仮想アドレス→物理アドレスの割り当ての解除
pmap_remove_all	仮想アドレス→物理アドレスの全割り当ての解除
pmap_activate	pmapを有効にする(プロセスの仮想メモリ空間切り替え)
pmap_deactivate	pmapを無効にする(プロセスの仮想メモリ空間切り替え)
pmap_unwire	WIREDの解除
pmap_protect	ページのread/write/executeの設定
pmap_page_protect	vm_pageのread/write/executeの設定
pmap_clear_modify	modifyedフラグのクリア
pmap_is_modified	modifiedフラグのチェック
pmap_clear_reference	referencedフラグのクリア
pmap_is_referenced	referencedフラグのチェック

- pmap_kenter_paとpmap_enterの違い
  - pmap_kenter_pa/pmap_kremove
    - 常にWIREDマッピング（vm_pageのことを考えなくてよい）
    - physical addressがvm_pageに紐付けされていない(かもしれない)
    - 割り込みコンテクストでも使える
  - pmap_enter/pmap_remove
    - WIREDじゃないかもしれない(flagsに従う)
    - managed mappingである(vm_pageを考慮する必要がある)
    - 既に存在する仮想アドレスに対して呼ばれることがある→変更として扱う
  - vm_pageとは?
    - 物理ページに紐つけられた構造。
      uvmが初期化時に作る。PHYS_TO_VM_PAGE(paddr) で取得可能
    - referenced/modifiedビットはここに保存される
    - pmap_page_protect等ではパラメータがvm_pageで渡される
      - 同じvm_page(物理ページ)に割り当てられた仮想ページのリストを持っておかなければいけない
        - » 物理アドレスがわかれば、割り当てている仮想アドレスのリストを取得できるようにしておく必要がある
      - 物理→仮想ページのリスト管理はpmap内で行う必要がある
        - » pmap_enter/pmap_removeでvm_pageのリストにvaddrを追加削除する必要がある
        - » リスト操作にロックが必要

vm_pageのロック

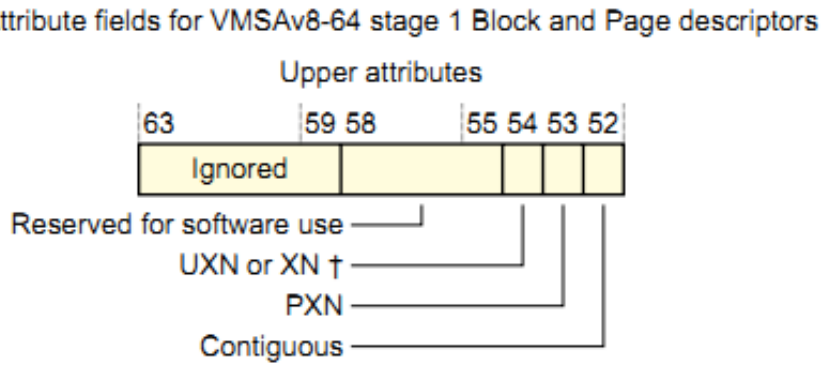
- vm_page内のリスト操作時にはロックが必要なため、aarch64では各vm_page毎にロックを持たせてみた
  → するとVM_MDPAGE_INIT時にmutexが溢れた
  （boot時に使えるmutexの最大数に達してしまった）
  - vm_pageは物理ページ数分存在する
  - VM_MDPAGE_INITを呼ぶ時点ではmallocができないためmutex領域を増やせない

しょうがないのでもう少し後でuvmから呼ばれる pmap_init() の中で自前で全vm_page分mutex_initするようにしてみた。

- mutexをvm_page別に分けた理由は、SMP時にロック競合が起きにくいと思われる為
  - まだよくわからない。今後のチューニング対象
  - ちなみにarmはグローバルにlockしている
  - x86はリストのポインタをhashしてlock?
  - そもそもaarch64でまだSMPが動いていないので（ry

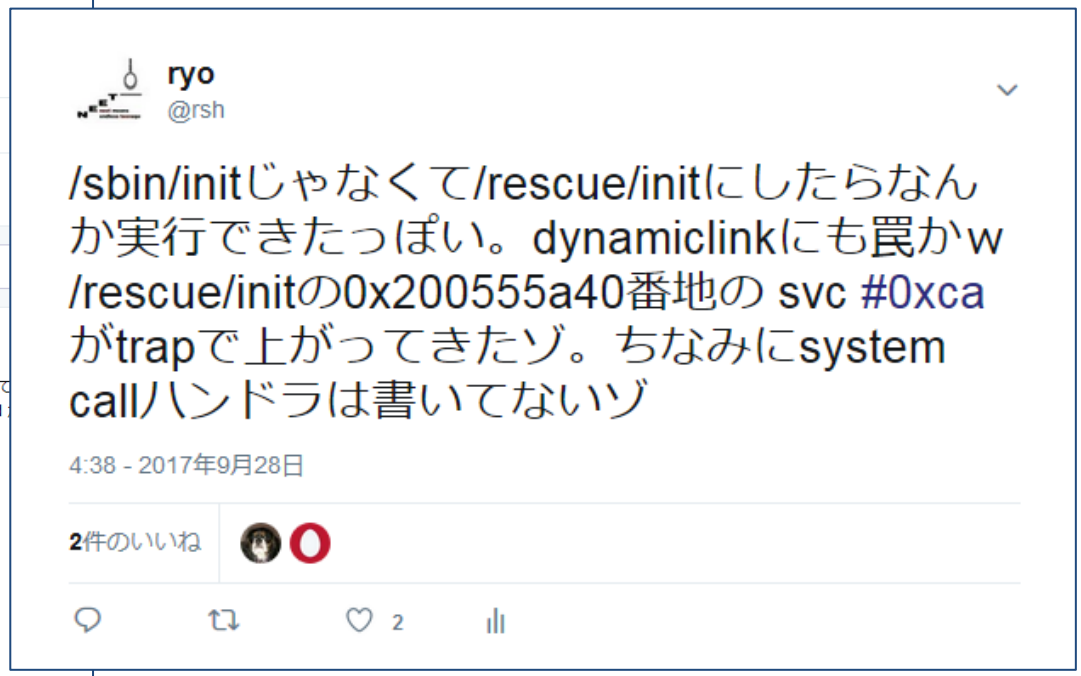
referenced/modified emulation

- pmap_enter時に設定された read/writeのフラグを、PTE(page table entry)の空きbitに設定しておく



- 実際にpteに設定するprotectのreadable/writableはクリアしておく→read/writeされるとpage faultが起こる
- read faultが起きたら本当にreadableかどうかチェック
  - readableでなければfault処理続行
  - そうでなければrefernced フラグを設定してreadableにしてreturn
- write faultが起きたらwritableかどうかチェック
  - writableでなければfault処理続行
  - そうでなければmodifiedフラグを設定してwritableにしてreturn

- pmapをこの通りに実装していくと、demand pagingが動き、プログラムの実行ができるようになる…？



- ちなみに svc #0xca は SYS_sysctl。

  forkして/rescue/initを読み込み、ユーザアドレス空間にスイッチ＆ジャンプして実行まではできたけど、システムコールで落ちたということです。

- この後、ddbを実装したり、たくさんのバグやcontext switchやsignalやsoftintなどとの長い戦いが続き…

ryo
@rsh

netbsd/evbarm64 multiuser works!

21:52 - 2017年11月30日

31件のリツイート 44件のいいね

1 31 44

俺たちの戦いはこれからだ！

完