# Solarisテクノロジーの魅力と活用術

～事例とデモで紹介！Solarisの問題解決テクノロジー～

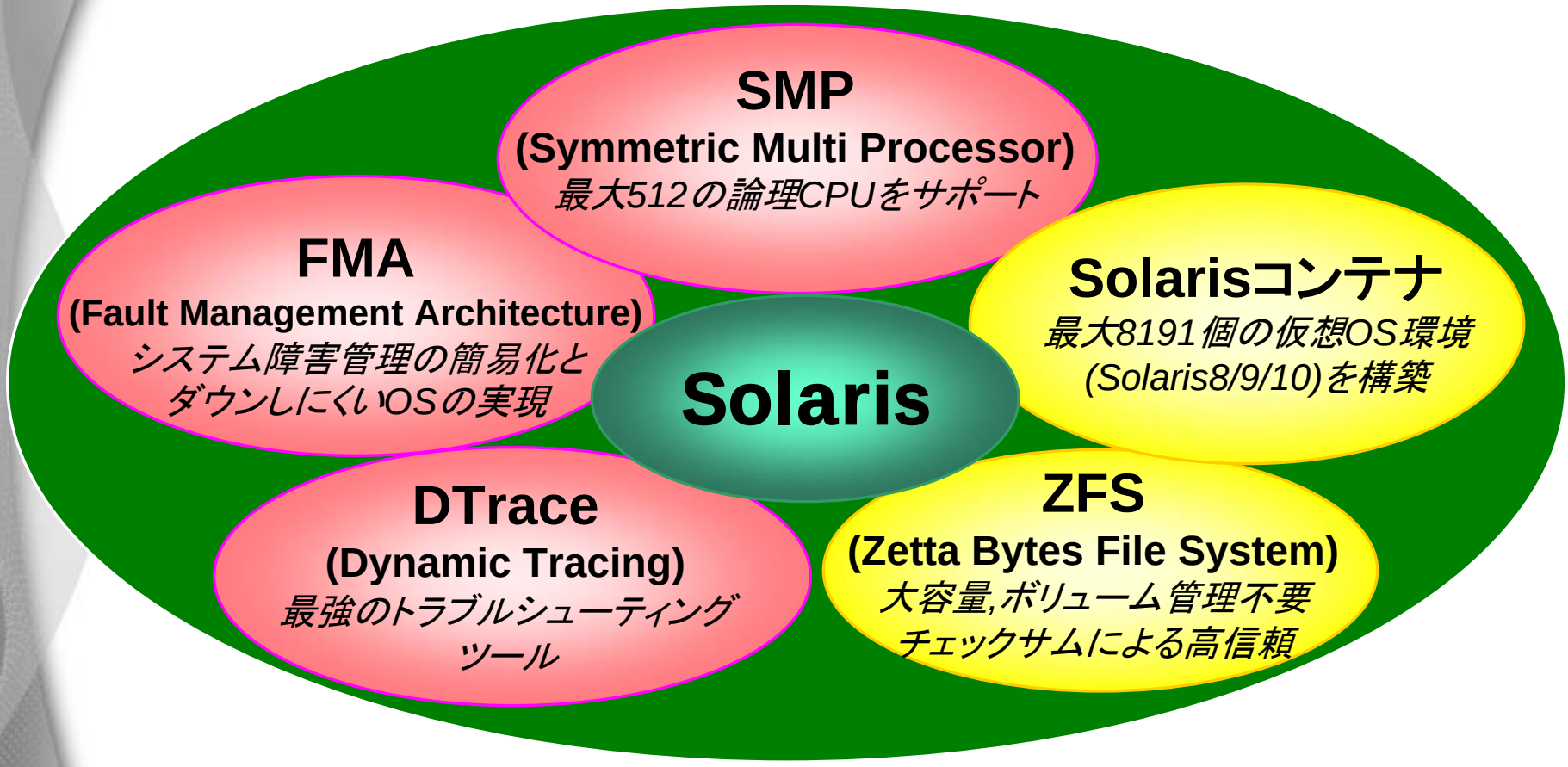
2008年11月6日

富士通株式会社

サーバシステム事業本部UNIXソフトウェア開発統括部
第一開発部長　東　圭三

プラットフォームソリューションセンター
プロダクトテクニカルセンター
プロジェクト課長　志賀　真之

# Solarisの新しいテクノロジー



# SMP
# (Symmetric Multi Processing)

*最大512の論理CPUをサポート*

# SMP OSとして業界トップクラスのCPU数をサポート

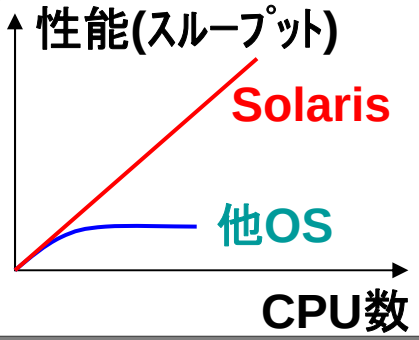
ソケット数
(物理CPU数)

コア数

スレッド数

| 論理CPU数 | 構成 | | | 機種名 | 発表 |
|---|---|---|---|---|---|
| 30 | 30 | 1 | 1 | Ultra Enterprise 6000 | 1996.04 |
| 64 | 64 | 1 | 1 | Sun Enterprise 10000 | 1997.01 |
| 128 | 128 | 1 | 1 | PRIMEPOWER 2000 | 2000.05 |
| 144 | 72 | 2 | 1 | Sun Fire E25K | 2004.02 |
| 256 | 64 | 2 | 2 | SPARC Enterprise M9000 (SPARC64-VI) | 2007.04 |
| 512 | 64 | 4 | 2 | SPARC Enterprise M9000 (SPARC64-VII) | 2008.07 |

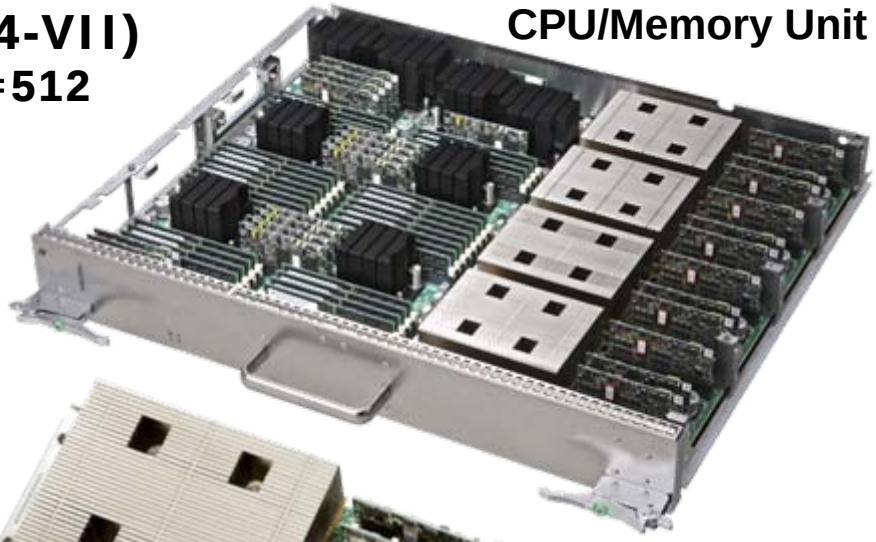
- いち早く、多CPUにおけるスケーラビリティを実現
- 他OSとは違い、CPU数が増えても、スループット性能がリニアに向上



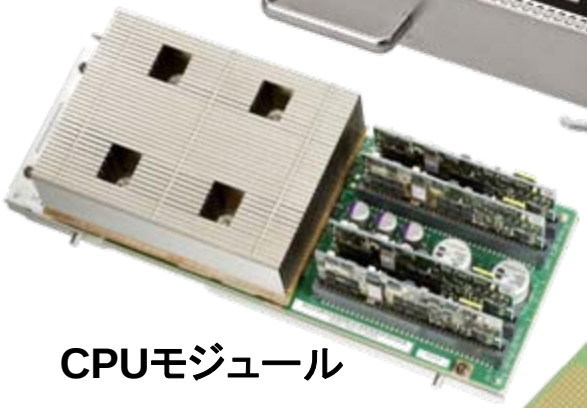
# 512論理CPUの世界を見てみよう(動画デモ:約3分)

**SPARC Enterprise M9000 (SPARC64-VII)**

- 最大 64 CPU×4 core×2 thread＝512
- 最大 2TBメモリ

CPU/Memory Unit



CPUモジュール


SPARC64 VII

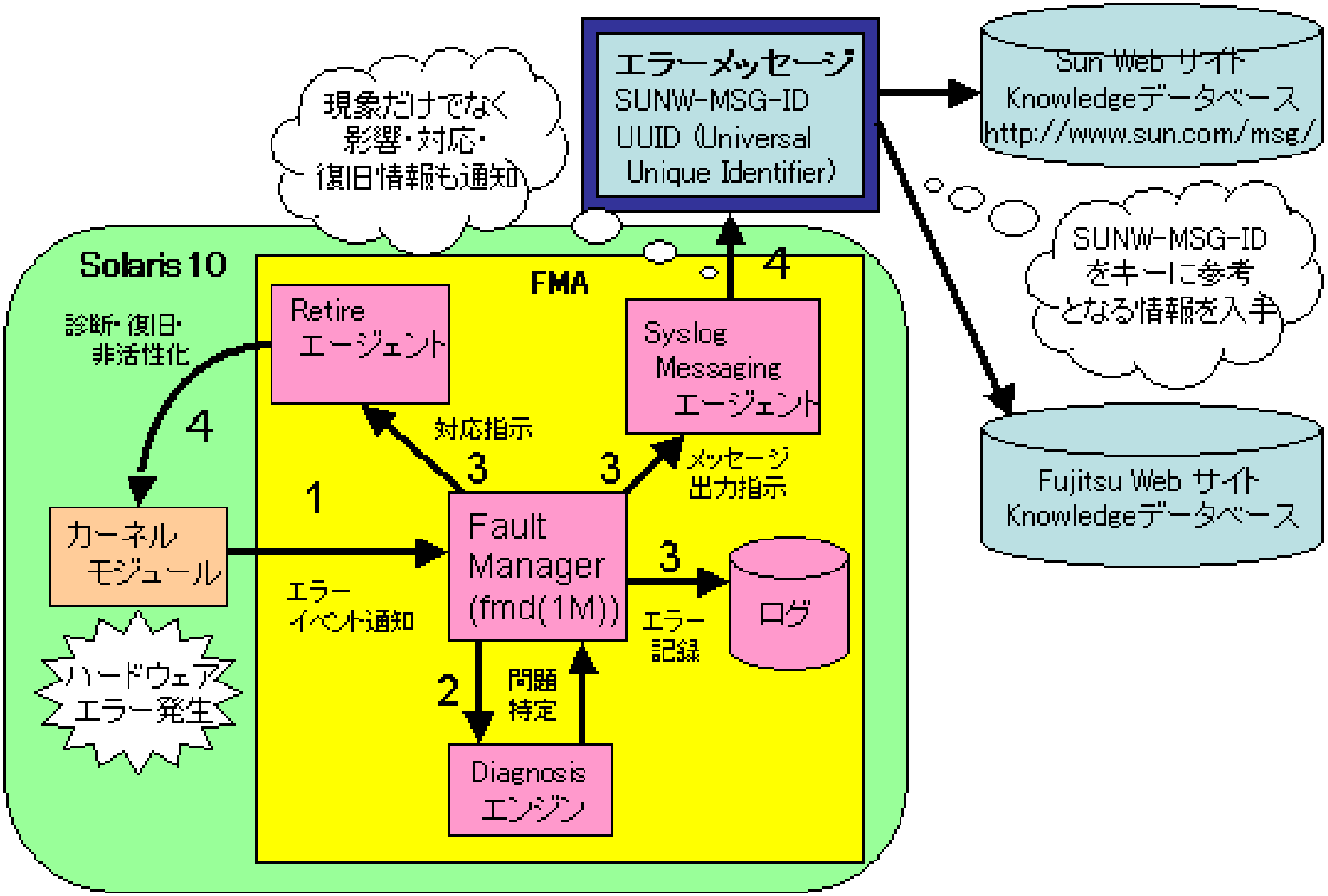
```
POST Sequence 0C Cacheable Instruction
POST Sequence 0D Softint
POST Sequence 0E CPU Cross Call
POST Sequence 0F CMU-CH
POST Sequence 10 PCI-CH
POST Sequence 11 Master Device
POST Sequence 12 DSCP
POST Sequence 13 SC Check Before STICK Diag
POST Sequence 14 STICK Stop
POST Sequence 15 STICK Start
POST Sequence 16 Error CPU Check
POST Sequence 17 System Configuration
POST Sequence 18 System Status Check
POST Sequence 19 System Status Check After Sync
POST Sequence 1A OpenBoot Start...
POST Sequence Complete.

Fujitsu SPARC Enterprise M9000 Server, using Domain console
Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
Copyright 2008 Sun Microsystems, Inc. and Fujitsu Limited. All rights res
OpenBoot 4.24.7, 1867776 MB memory installed, Serial #9568600.
Ethernet address 0:b:5d:dc:1:58, Host ID: 80920158.



Aborting auto-boot sequence.
[0] ok
```

# FMA

## (Fault Management Architecture)

*システム障害管理の簡易化と*
*ダウンしにくいOSの実現*

# FMA (Fault Management Architecture)とは



# FMAにより、このようなことが良くなりました

## システム故障の管理が簡単に

- fmadmコマンドで故障箇所と交換部品を表示
- 表示されたURLを見れば、詳細や対処方法が一目瞭然

## ハード故障時でも極力システムダウンさせない

- メモリやCPU、I/Oの多bitエラーでも、なるべく運用継続
- エラーにより終了したプロセスを自動的に再起動
- 故障箇所を自動的に切り離し

## 今は影響がない障害でも、予測して自動的に対処

- 全メモリを走査して、エラーの有無を確認
- 多bitエラーは即時、1bitエラーは多発時に切り離し

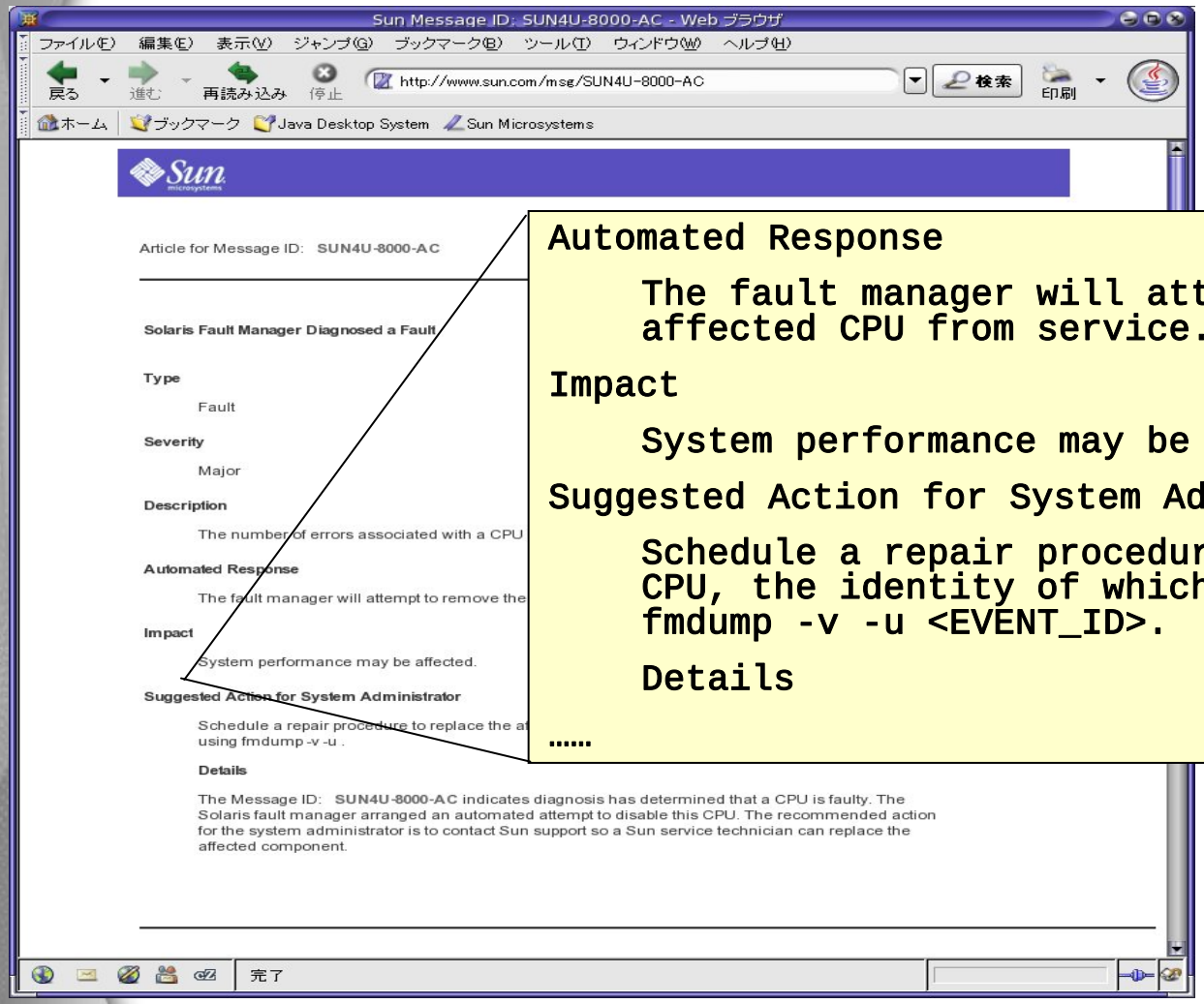
# fmadmコマンドにより交換部品がすぐに判明

```
# fmadm faulty
--------------- ------------------------------------  -------------- ---------
TIME            EVENT-ID                              MSG-ID         SEVERITY
--------------- ------------------------------------  -------------- ---------
Oct 10 22:17:47 977053d0-a5ab-c208-865e-ddedd7b5abd2  SUN4U-8000-2S  Major

Fault class : fault.memory.dimm 95%
Affects     : mem:///unum=/MBU_A/MEM3B
                  degraded but still in service
FRU         : mem:///unum=/MBU_A/MEM3B 95%
                  faulty
Serial ID.  : F422F122:M3 93T2950CZ3-CD5

Description : The number of errors associated with this memory module has
              exceeded acceptable levels.  Refer to
              http://sun.com/msg/SUN4U-8000-2S for more information.

Response    : Pages of memory associated with this memory module are being
              removed from service as errors are reported.

Impact      : Total system memory capacity will be reduced as pages are
              retired.

Action      : Schedule a repair procedure to replace the affected memory
              module. Use fmdump -v -u <EVENT_ID> to identify the module.
```

FRU (Field Replaceable Unit) に、交換部品の情報が表示される

# Knowledge Article Webサイト

## エラーの詳細や対処方法を表示されたURLで確認




```
Automated Response
    The fault manager will attempt to remove the
    affected CPU from service.
Impact
    System performance may be affected.
Suggested Action for System Administrator
    Schedule a repair procedure to replace the affected
    CPU, the identity of which can be determined using
    fmdump -v -u <EVENT_ID>.
    Details
……
```

# メモリの多bitエラーでも再起動は不要 (ユーザ空間の場合)

## ユーザ空間でメモリのECC多bitエラーが発生した場合

- 太古のSolarisおよび他OS
  パニックしてリブート
- Solaris9以前
  発生したプログラムを停止し、システムを自動的に再起動

メモリの大容量化に伴い、お客様からは不評

Solaris 10 では…

- システムの再起動なし
- 発生したプログラムを停止
- さらにSMFで登録されているプログラムなら自動的に再実行

# DTrace

## (Dynamic Tracing)

*最強のトラブルシューティングツール*

# 今までのトラブルシューティング

ダンプ解析

カーネルダンプやプロセスダンプの解析

本稼動中だよ！
システム止めないで！

いろいろなツールはあるが･･･

sar, truss, vmstat,

mpstat, lockstat, cpc, ..

調べたい所になかなかジャストヒットしない
trussは便利だけど重い

こんなツールもあるが

audit, libumem,

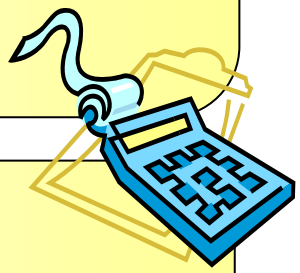
watchmalloc, ..

再起動が必要だけど
本稼動中なので許可出ない

# DTraceはこのように便利です！

## 稼働中のまま、使用できます

- プログラムの再コンパイルや再起動が不要
- システムの再起動が不要
- 観測点を絞ればオーバーヘッド小



## システム全体を観測できます

- ユーザプログラムだけでなく、カーネルも観測可能
- 約 70,000 もの観測ポイント

## わかりやすいスクリプト言語“D”で記述します

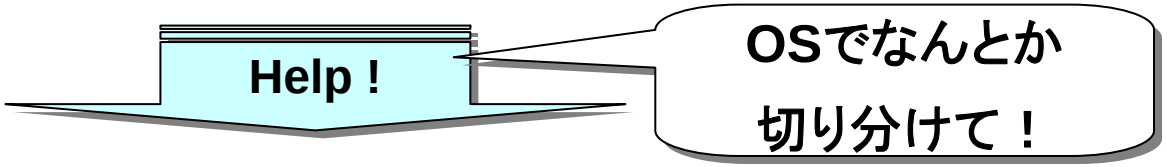
- C言語と類似、awkライクな機能
- コマンドラインでの実行も可能

# 調査事例① 誰がメモリリークしている？

プログラムのプロセスサイズが巨大化していくトラブルが発生
よくあるメモリリーク障害、以下の調査ツールを紹介
libumem, watchmalloc, Purify

しかし本稼動中のため、お客様は拒否
- 再コンパイル、起動スクリプトの変更、再起動は禁止
- 実行オーバーヘッドが大きすぎる

いろんなライブラリを結合したプログラム
⇒開発者多数のため、ソースコードから犯人を絞り込めず




DTrace を使って見事解決

# どんなスクリプトで調べたか見てましょう

brk(2)、mmap(2)が発行されたタイミングでスタックトースを表示するDスクリプトを実行して原因を特定

```
# cat brk.d
syscall::brk*:entry
/curpsinfo->pr_fname == "sample"/
{
        printf("¥n PID=%d, name=%s, endds=0x%lx",
                pid, curpsinfo->pr_fname, arg0);
        ustack();
}

syscall::mmap*:entry
/curpsinfo->pr_fname == "sample"/
{
        printf("¥n PID=%d, name=%s, addr=0x%lx, size=0x%lx",
                pid, curpsinfo->pr_fname, arg0, arg1);
        ustack();
}
```

観測対象をsampleというプロセスに限定
⇒オーバーヘッド小

# このような結果が表示されました

```
# dtrace -s brk.d
dtrace: script 'brk.d' matched 3 probes
CPU     ID                    FUNCTION:NAME
  0   2109                        brk:entry
 PID=22482, name=sample, endds=0x21c48
              libc.so.1`_brk_unlocked+0x4
              libc.so.1`sbrk+0x24
              libc.so.1`_morecore+0x24
              libc.so.1`_malloc_unlocked+0x1fc
              libc.so.1`_smalloc+0x4c
              libc.so.1`malloc+0x4c
              sample`func1+0x4
              libc.so.1`_lwp_start
```

コマンドラインで実行可

func1という関数がmalloc を呼び出している。これが犯人

# 調査事例② 誰が私のプロセスをkillした？

複数のメーカの複数の製品で構築したシステムで、
killされるはずのないプロセスが突然シグナルを受けてアボート
⇒犯人探しするも、どの製品も「私は白です」「そんなことしません」

Solaris9までは良い調査ツールなく、audit(監査機能)を利用。
しかし多くのお客様はauditの利用を嫌がった。

- auditを有効化するには再起動が必要
- 大量の監査ログが採取される。オーバーヘッドも心配。

Help！

OSでなんとか
切り分けて！

DTrace を使って見事解決

類似例として“誰が私のIPC資源を勝手に削除した？”
“誰が私のファイルを勝手に削除した？”にも効果を発揮

# どんなスクリプトで調べたか見てましょう

観測対象をsampleというプロセスに限定
⇒オーバーヘッド小

```
# cat signal.d
fbt::sigtoproc:entry
/args[0]->p_user.u_comm == “sample"/
{
        printf(“¥n SENDER: PID=%d, name=%s",
               pid, curpsinfo->pr_fname);
        printf("¥n RECIPIENT: PID=%d, proc=%x(%s), sig=%d",
               args[0]->p_pidp->pid_id, arg0,
               args[0]->p_user.u_comm, arg2);
        stack();    ★カーネル空間のスタックトレースを表示
        ustack();   ★ユーザ空間のスタックトレースを表示
}
```

# このような結果が表示されました

```
# dtrace -s signal.d
dtrace: script 'signal.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  2  14787                  sigtoproc:entry
 SENDER: PID=24347, name=tcsh
 RECIPIENT: PID=538, proc=30008b4e790(sample), sig=9
```

コマンドラインで実行可

★カーネル空間のスタックトレース

```
genunix`sigsendproc+0x1f4
genunix`sigqkill+0x94
genunix`kill+0x28
unix`syscall_trap32+0xcc
```

★ユーザ空間のスタックトレース

```
libc.so.1`kill+0x4
tcsh`execute+0x860
tcsh`execute+0xa7c
tcsh`process+0x57c
tcsh`main+0x22a8
tcsh`_start+0x108
```

tcshプロセス(PID=24347)がsampleプロセス(PID=538)にSIGKILL(sig=9) を送信。これが犯人。

# このような結果が表示された場合もありました

```
# dtrace -s signal.d
dtrace: script 'signal.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  3  17954                  sigtoproc:entry
 SENDER : PID=0, name=sched
 RECIPIENT : PID=538, proc=30004e8d370(sample), sig=2
★カーネル空間の genunix`pgsignal+0x34
 スタックトレース genunix`strsignal+0x5c
                genunix`strrput_nondata+0x5fc
                genunix`strrput+0x4c8
                unix`putnext+0x218
                unix`putnext+0x218
                genunix`putnextctl1+0x74
                ldterm`ldterm_dosig+0x154
                ldterm`ldtermrput+0x7cc
                unix`putnext+0x218
                unix`putnext+0x218
                ptm`ptmwsrv+0xc8
                genunix`runservice+0x40
                genunix`stream_service+0x5c
                genunix`taskq_d_thread+0x88
                unix`thread_start+0x4
```

コマンドラインで実行可

シグナルの送信元はカーネル内の ldtermモジュール
Ctl-Cを押したことにより、sampleプロセス(PID=538)が SIGINT(sig=2) を受信。

# Dtraceを試してみよう

- 便利なサンプルスクリプト
- サンプルを試してみよう
- スクリプトの構造
- 出力をカスタマイズ
- コマンドラインで簡単に使う例
- Dtrace活用例

# 便利なサンプルスクリプト

- Solaris10をインストールするとサンプルスクリプトがインストールされます。 /usr/demo/dtrace
- インストールされているスクリプト

| | | | | | |
|---|---|---|---|---|---|
| applicat.d | find.d | ksyms.d | qtime.d | ticktime.d | whoqueue.d |
| badopen.d | firebird.d | libc.d | renormalize.d | time.d | whosteal.d |
| begin.d | hello.d | lquantize.d | restest.d | tracewrite.d | whowrite.d |
| callout.d | howlong.d | lwptime.d | ring.d | trunc.d | writes.d |
| clause.d | index.html | normalize.d | rtime.d | trussrw.d | writesbycmd.d |
| clear.d | interp.d | nscd.d | rwinfo.d | userfunc.d | writesbycmdfd.d |
| countdown.d | interval.d | pri.d | rwtime.d | whatfor.d | writetime.d |
| counter.d | intr.d | printa.d | sig.d | whatlock.d | writetimeq.d |
| dateprof.d | iocpu.d | pritime.d | soffice.d | where.d | xioctl.d |
| delay.d | iosnoop.d | prof.d | spec.d | whererun.d | xterm.d |
| denorm.d | iothrough.d | profpri.d | specopen.d | whoexec.d | xwork.d |
| end.d | iotime.d | progtime.d | ssd.d | whofor.d | |
| error.d | iprb.d | putnext.d | syscall.d | whoio.d | |
| errorpath.d | kstat.d | qlen.d | tick.d | whopreempt.d | |

- 日本語ドキュメントはここで入手します
http://docs.sun.com/
**Solaris 10 Software Developer Collection – Japanese / DTrace ユーザーガイド**

# サンプルを試してみよう(1)

- 実行方法
  dtrace –s <ファイル名>　で実行します
- お決まりの hello.d

```
# dtrace -s hello.d
dtrace: script 'hello.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  1      1                           :BEGIN   hello, world
```

- whoexec.d
  どこからプロセス起動したのかがわかる

```
pw250s10# dtrace -s whoexec.d
^C
WHO                  WHAT                 COUNT
chkstatus            grep                 1
chkstatus            sed                  1
chkstatus            setmadmdb            1
clear                tput                 1
csh                  clear                1
csh                  du                   1
csh                  ls                   1
du                   du                   1
```

# サンプルを試してみよう(2)

- progtime.d
  プログラムの実行時間の統計を表示する

```
w250s10# dtrace -s progtime.d
dtrace: script 'progtime.d' matched 2 probes

  cat
           value  ------------- Distribution ------------- count
         1048576 |                                         0
         2097152 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2
         4194304 |                                         0

  grep
           value  ------------- Distribution ------------- count
         1048576 |                                         0
         2097152 |@@@@@@@@@@@@@@@@@@@@                     1
         4194304 |@@@@@@@@@@@@@@@@@@@@                     1
         8388608 |                                         0

 sh
           value  ------------- Distribution ------------- count
         1048576 |                                         0
         2097152 |@@@@                                     1
         4194304 |@@@@@@@@@@@                              3
         8388608 |@@@@@@@                                  2
        16777216 |@@@@@@@@@@@@@@@@@@                       5
        33554432 |                                         0
```

# サンプルを試してみよう(3)

- writetime.d
  write システムコールにかかった時間を平均を表示

```
pw250s10# dtrace -s writetime.d
dtrace: script 'writetime.d' matched 2 probes
^C
  date                                                          34900
  sed                                                           35814
  sac                                                           35850
  pollstat                                                      41985
  dtrace                                                        44300
  init                                                          45686
  inetd                                                         52950
  syslogd                                                       55454
  java                                                          60830
  cron                                                          72883
  sadc                                                          76929
  mail.local                                                    89635
  sendmail                                                     107998
  imapd                                                        169146
```

# スクリプトの構造

- writetime.d
  write システムコールにかかった時間の平均を表示

**[例1]write(2)システムコールの実行時間の平均値をコマンドごとに表示する。**

```
/*  writetime.d */
syscall::write:entry
{
    self->ts = timestamp;
}

syscall::write:return
{
    @time[execname] = avg(timestamp – self->ts);
    self->ts = 0;
}
```

“self->ts”という書式は、「スレッドローカルな変数」を示すもの。プローブは異なるスレッドで同時に呼び出される可能性があるが、“self->変数名（任意）”という書式を使うと、スレッド毎に別々の変数が用意されるため、値が混じらなくなる。

“timestamp”はbootしてからの時間を示す（単位：nano-second）

**性能関連の調査をする場合、ある一定期間のデータを収集し、分析する。DTrace の Aggregation と呼ばれる機能（集積関数）はそのような調査に対して有効に利用できる。**

```
@name[ keys ] = aggfunc ( args ) ;

name     : ユーザが定義する Aggregation の識別子
           （省略可能）
keys     : keys で指定した項目ごとに集計する
aggfunc  : dtrace に用意された集積関数
args     : aggfunc の引数
```

# 出力をカスタマイズ

**[例1]write(2)システムコールの実行時間の平均値をコマンドごとに表示する。**

```
/*  writetime.d */
syscall::write:entry
{
    self->ts = timestamp;
}

syscall::write:return
{
    @time[execname] = avg(timestamp – self->ts);
    self->ts = 0;
}
```

変更前

**[例2]write(2)システムコールにかかった時間の分布をコマンド毎に表示する。**

```
/*  quantize.d */
syscall::write:entry
{
    self->ts = timestamp;
}

syscall::write:return
{
    @time[execname] = quantize(timestamp – self->ts);
    self->ts = 0;
}
```

変更点

実行結果

```
# dtrace -s writetime.d
dtrace: script './writetime.d' matched 2 probes
^C
  iropt                                          31315
  acomp                                          37037
  make.bin                                       63736
  tee                                            68702
  date                                           84020
  sh                                             91632
  dtrace                                        159200
                    ....
```

実行結果

```
# dtrace -q -s quantize.d
  acomp
  value   ----- Distribution ----------- count
  4096    |                                  0
  8192    |@@@@@@@@@@@@                      840
  16384   |@@@@@@@@@@@                       750
  32768   |@@                                165
  65536   |@@@@@@                            460
  131072  |@@@@@@                            446
  262144  |                                  16
  524288  |                                  0
  1048576 |                                  1
  2097152 |                                  0
```

# コマンドラインで簡単に使う例

- システムコールを発行しているプログラム名とCall数を表示

```
# dtrace -n sysexec'{@[execname] = count()}'
dtrace: description 'sysexec' matched 1 probe
^C
sh                                                          1
hardmond                                                    2
chkstatus                                                   3
```

- スレッドを起動したプログラム名と起動数を表示

```
# dtrace -n nthreads'{@[execname] = count()}'
dtrace: description 'nthreads' matched 1 probe
^C
 java                                                       2
 pollstat                                                   2
 sendmail                                                   2
 setrci                                                     2
 chkstatus                                                  3
 sh                                                         53
```

# コマンドラインで簡単に使う例

- クロスコールを発行しているプログラム名とCall数を表示

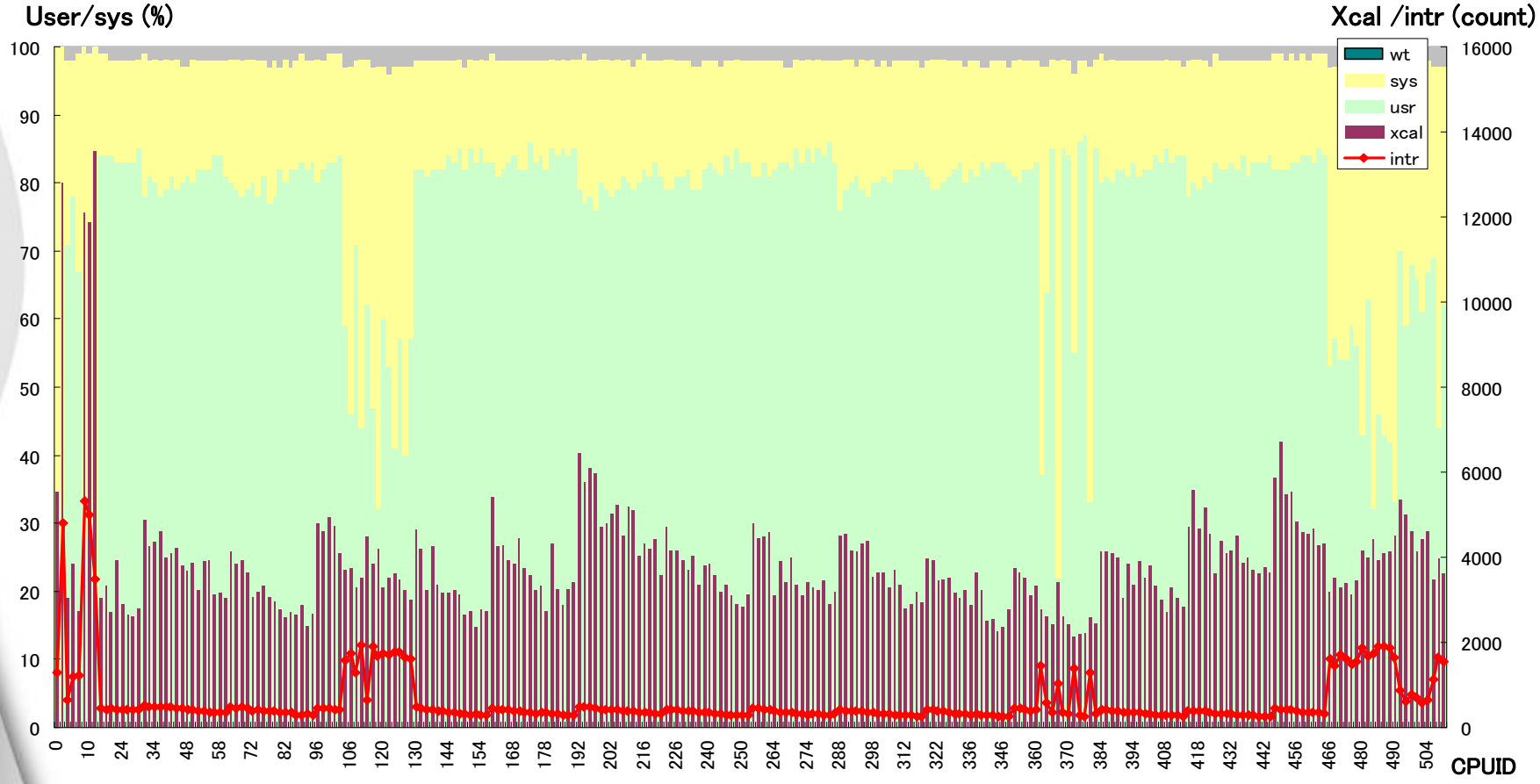
```
# dtrace -n xcalls'{@[execname] = count()}'
dtrace: description 'xcalls' matched 4 probes
^C
snmpd                                                          2
fsflush                                                        8
java                                                          11
ps                                                            17
```

- トラップを発生させたプログラム名とトラップ回数を表示

```
# dtrace -n trap'{@[execname] = count()}'
dtrace: description 'trap' matched 2 probes
^C
pipeopener                                                   261
getmodelcode                                                 280
setrci                                                       312
prtmadmlog                                                   314
smbd                                                         428
mail.local                                                   439
```

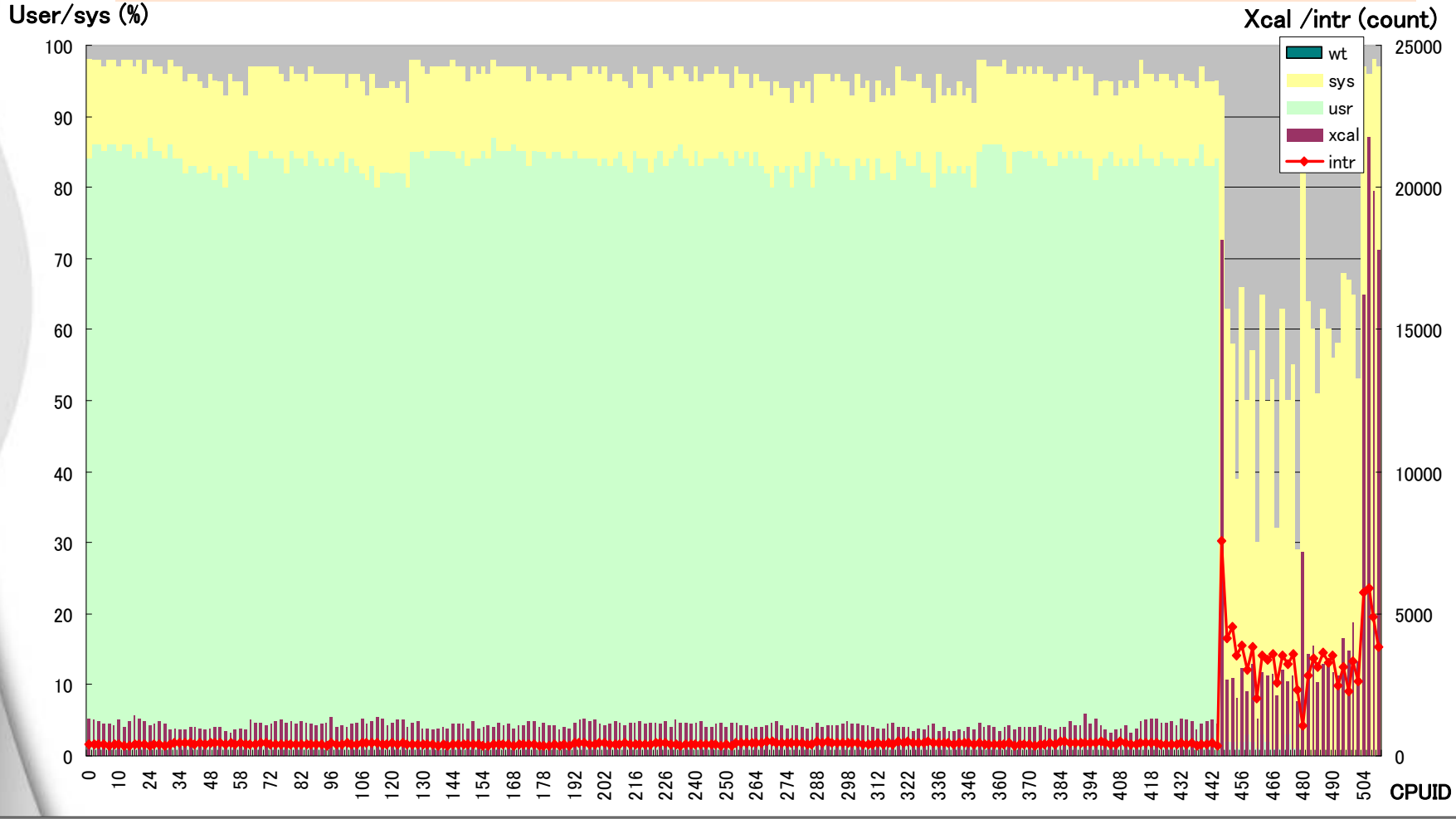
# Dtrace活用例（チューニング前）

Oracleのチューニングにより、CPUを使い切った状態
mpstatでxcalが多かったので、Dtraceで原因を調査し、
xcallの多くがI/Oの処理に依存していることを確認



# Dtrace活用例（チューニング後）

プロセッサセットを作ってOracleのshadowプロセスをバインドし、
I/Oの処理でOracleのshadowプロセスを乱さないように処置
DBのスループット性能が約2倍に改善。



FUJITSU

shaping tomorrow with you