ipcmd:

A command-line interface to SysV semaphores and message queues

Nathan Weeks
M.S. Computer Science candidate

April 10, 2012
Creative Component

Introduction

What is ipcmd?

- An easy-to-use command-line interface to SysV semaphores and message queues
  - SysV shared memory not supported due to the limited utility of a command-line interface
- Allows application developers to easily prototype, debug, and test the use of SysV semaphores and message queues in applications
- Provides a high-level, scriptable interface to system administrators/script writers for coordinating concurrent processes
- Open source (BSD license): http://ipcmd.googlecode.com/

Outline

SysV IPC

- Consists of message queues, semaphores, and shared memory
  - Enable inter-process communication (IPC) within a single operating system instance; not distributed
- Developed in the mid 1970's by Bell Labs for Columbus UNIX
- Released outside Bell Labs in System V UNIX (1983)
- Standardized by POSIX ("XSI IPC")
- Available on practically all non-embedded *nix systems
- Used by a number of applications:
  - Database (Oracle, PostgreSQL, Firebird)
  - Web (Apache, TYPO3, Google Chrome)
  - Middleware (MPICH-1, Open MPI, WebSphere MQ)

Message Queues

- Enable asynchronous inter-process communication
- Sender blocks if there is not enough space in the queue to contain the message [1]
- Sender assigns an integer type > 0 to each message.
- Receiver selects messages with *msgrcv(...,msgtyp,...)*:
  - if $msgtyp = 0$, receive first message on the queue
  - if $msgtyp > 0$, receive first message of type *msgtyp*
  - if $msgtyp < 0$, receive first message of the lowest type that is $\leq |msgtyp|$
- Receiver will block if a message of the appropriate type does not exist

[1] Default message queue size ranges from 2K on BSD/OS X to 64K on Linux and Solaris

Semaphores

Definition

(Dijkstra, 1965) A *counting semaphore* is a nonnegative integer variable S that can be modified by two atomic operations:
$V(S)$: $S \leftarrow S + 1$
$P(S)$: suspends until $S > 0$, then $S \leftarrow S - 1$.

SysV semaphores extend this definition:

- Three operations are defined:
  - Increase the value of a semaphore by an integral amount
  - Decrease the value of a semaphore by an integral amount
  - Wait for a semaphore value to become 0
- An array of semaphore operations may be performed on a set of semaphores. The caller will block until the all semaphore operations can be completed in array order.

What's Wrong with SysV IPC?

Michael Kerrisk, *The Linux Programming Interface*:

> *The programming interface provided by System V is overly complex.*

C. W. Higginbotham and R. Morelli, *A System for Teaching Concurrent Programming*:

> *. . . direct use of Unix semaphore system calls is hopelessly complicated and far too messy with system-dependent details to be of much use in the classroom.*

Marc J. Rochkind, *Advanced Unix Programming, Second Edition*

> *For interprocess communication, the System V semaphore system calls are ridiculously difficult to use and overwrought with features, but they're universally available and not so bad once the hard stuff (initialization, mainly) is hidden behind a reasonable interface*

How Can ipcmd Help?

The effect of the C program:

```c
#define _XOPEN_SOURCE 600
#include <stdlib.h>
#include <sys/sem.h>

int main(void) {
    size_t nsops;
    struct sembuf *sops;

    int semid = strtol(getenv("IPCMD_SEMID"), NULL, 10);

    short sem_op = 1;

    union semun {
        int val;
        struct semid_ds *buf;
        unsigned short  *array;
    } arg;
    struct semid_ds seminfo;

    arg.buf = &seminfo;

    semctl(semid, 0, IPC_STAT, arg);

    nsops = (size_t)arg.buf->sem_nsems;

    sops=(struct sembuf *)malloc(nsops*sizeof(struct sembuf));
    for (unsigned short sem_num=0; sem_num < nsops; sem_num++) {
       sops[sem_num].sem_num = sem_num;
       sops[sem_num].sem_op = sem_op;
       sops[sem_num].sem_flg = 0;
    }

    semop(semid, sops, nsops);
}
```

is equivalent[2] to the command:

$ ipcmd semop 1

[2] The C example neglects the numerous error checks done by ipcmd

Application Developers

Application developers that already use SysV semaphores/message queues can use ipcmd for several purposes:

- Prototyping
  - Protype application in scripting language, or
  - Invoke ipcmd directly from system/application programming language
- Debugging
  - For example, query semaphore values for a deadlocked application
- Testing
  - For example, initialize semaphores to different states to test various scenarios

System Administrators

System administrators (or developers that use scripting languages) can use ipcmd to coordinate the execution of concurrent processes. Examples include:

- Serializing critical sections in makefiles
- Accelerating Unix pipelines by replicating filter processes
- Limiting the number of concurrent child processes (like *xargs -P*, but more flexible)

Overview

ipcmd is a single executable that implements the following subcommands:

```
$ ipcmd
usage: ipcmd <command> [options] [args]

Where <command> is one of the following:
    ftok        generate an IPC key
    msgget      create a message queue
    msgrcv      receive a message
    msgsnd      send a message
    semctl      initialization/query semaphores
    semget      create a semaphore set
    semop       semaphore operations
```

Message Queues

ipcmd implements the SysV message queue API thus:

- ipcmd msgget [-Q msgkey [-e]] [-m mode]
- ipcmd msgsnd [-q msqid] [-t mtype] [-n] [message...]
- ipcmd msgrcv [-q msqid] [-t msgtyp] [-n] [-v]

Example

```
$ export IPCMD_MSQID=$(ipcmd msgget)   # IPC_PRIVATE is default
$ ipcmd msgsnd -t 2 $'message 1\n'     # message argument
$ echo 'message 2' | ipcmd msgsnd -t 1 # read message from stdin
$ ipcmd msgrcv -t 1
message 2
$ ipcmd msgrcv    # -t 0 is default (first message of any type)
message 1
$ ipcrm -q $IPCMD_MSQID                # remove message queue
```

Semaphores

ipcmd accepts the following semaphore options / arguments:

- ipcmd semget [-S semkey [-e]] [-m mode] [-N nsems]
- ipcmd semop [-s semid] [-n] [-u] sem_op [: command [argument...]]
- ipcmd semop [-s semid] [-n] [-u] sem_num_lbound[:sem_num_ubound]=sem_op[n][u]...[: command [argument...]]
- ipcmd semctl [-s semid] getval SEMNUM
- ipcmd semctl [-s semid] setval SEMNUM SEMVAL
- ipcmd semctl [-s semid] getpid SEMNUM
- ipcmd semctl [-s semid] getncnt SEMNUM
- ipcmd semctl [-s semid] getzcnt SEMNUM
- ipcmd semctl [-s semid] getall
- ipcmd semctl [-s semid] setall [SEMNUM_LBOUND[,SEMNUM_UBOUND]=]SEMVAL...

Semaphores

Examples

```
# create a set of 4 semaphores
$ export IPCMD_SEMID=$(ipcmd semget -N 4)
$ ipcmd semctl setall 1 # initialize all semaphores to 1
$ ipcmd semop 0:2=-1 3=+1
$ ipcmd semctl getall
0 0 0 2
$ ipcrm -s $IPCMD_SEMID # remove semaphore set
```

Dining Philosophers (Dijkstra, 1965)

- Five philosophers sit around table, each with a plate of noodles in front of them
- A chopstick is placed in between each philosopher
- A philosopher is either thinking or eating
- To eat, a philosopher must grab both the chopstick to their left and the chopstick to their right
- When a philosopher is done eating, they place both chopsticks back on the table and resume thinking

Deadlock occurs if each philosopher picks up their left (right) chopstick and perpetually wait for the right (left) chopstick to become available

Starvation results if a philosopher is never able to obtain both chopsticks

Dining Philosophers (Solution)

Solution using SysV semaphores:

1. Create a set of 5 semaphores (one for each chopstick)
2. Initialize the semaphores to 1
3. To eat, the philosopher performs an array of two semaphore operations: acquire (decrement) the left semaphore, acquire the right semaphore
4. After eating for a random number of seconds, a philosopher releases (increments) the left and right semaphores, then thinks for a random number of seconds

- Avoids *deadlock*: both chopsticks are obtained in one atomic operation
- Avoids *starvation* due to random intervals of eating/thinking

Dining Philosophers (code) I

```
philosopher() {
  left_chopstick=$((rank-1 < 0 ? NUM_PHILOSOPHERS-1 : rank-1))
  right_chopstick=$rank
  for ((meal=1 ; meal <= NUM_MEALS ; meal++))
  do
    echo "Philosopher $rank is hungry... reaching for chopsticks"
    ipcmd semop $left_chopstick=-1 $right_chopstick=-1
    echo "Philosopher $rank is eating meal $meal"
    sleep $((RANDOM%3)) # eat for 0-2 seconds
    echo "Philosopher $rank is done eating... thinking"
    ipcmd semop $left_chopstick=+1 $right_chopstick=+1
    sleep $((RANDOM%3)) # think for 0-2 seconds
  done
}

readonly NUM_PHILOSOPHERS=5 NUM_MEALS=3

# create a set of $NUM_PHILOSOPHERS semaphores
export IPCMD_SEMID=$(ipcmd semget -N $NUM_PHILOSOPHERS)
```

Dining Philosophers (code) II

```
# remove the semaphore set when this script exits
trap 'ipcrm -s $IPCMD_SEMID' EXIT

# place one chopstick between each philosopher
ipcmd semctl setall 1

# start $NUM_PHILOSOPHERS philosopher processes
for ((rank=0 ; rank < NUM_PHILOSOPHERS ; rank++))
do
  philosopher &        # function defined in next slide
done

# wait for background philosopher processes to terminate
# before exiting & removing the semaphore set
wait
```

make

Many variants of *make* support parallel compilation via a *-j* option. However, the GNU make manual contains the following caveat:

11.3 Dangers When Using Static Archives

It is important to be careful when using parallel execution (the -j switch...) and archives. If multiple ar commands run at the same time on the same archive file, they will not know about each other and can corrupt the file.
Possibly a future version of make will provide a mechanism to circumvent this problem by serializing all recipes that operate on the same archive file. But for the time being, you must either write your makefiles to avoid this problem in some other way, or not use -j.

make

The MPICH2 FAQ contains the following entry:

Q: The build fails when I use parallel make

A: *Parallel make (often invoked with make -j4) will cause several job steps in the build process to update the same library file (libmpich.a) concurrently. Unfortunately, neither the ar nor the ranlib programs correctly handle this case, and the result is a corrupted library. For now, the solution is to not use a parallel make when building MPICH2.*[a]

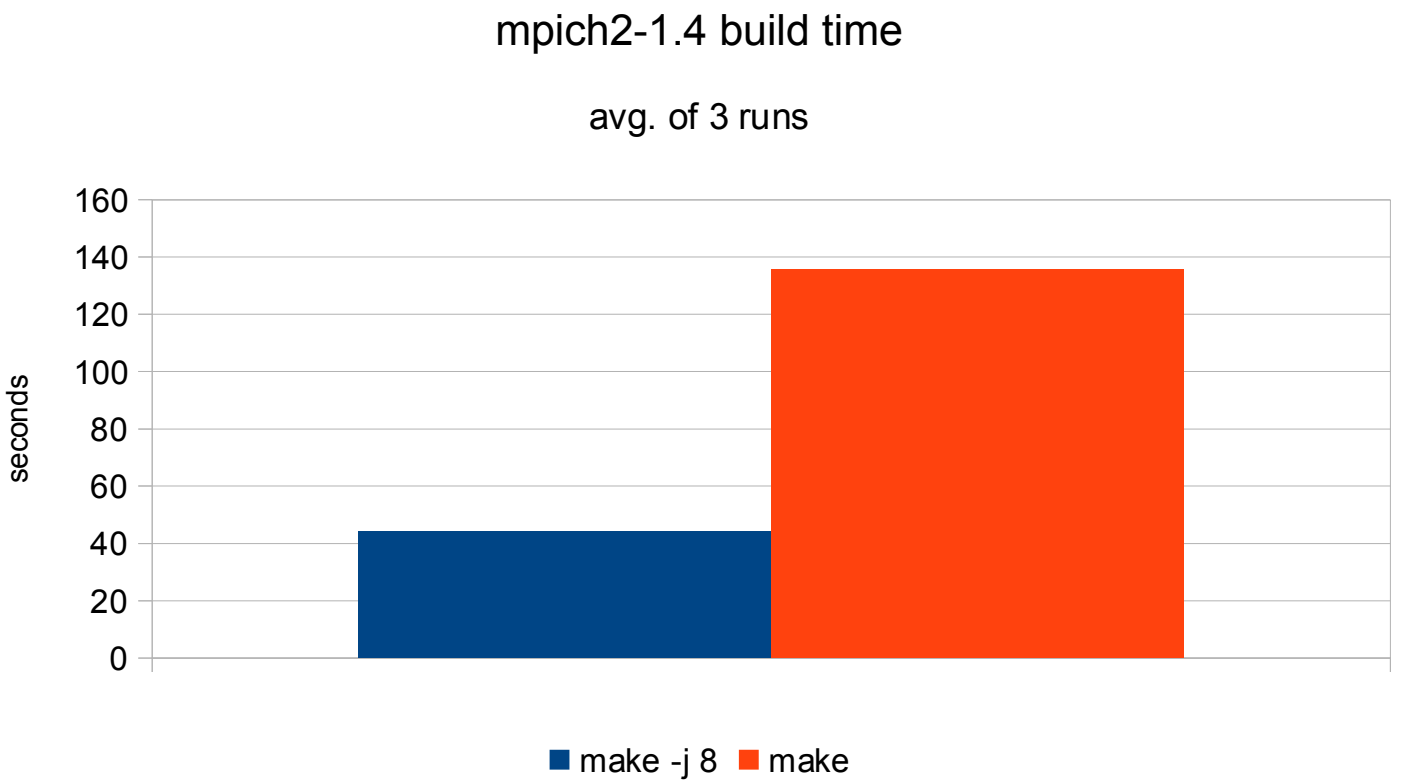
[a] This issue is fixed in the forthcoming MPICH2-1.5.

make

Solution: Use a semaphore to serialize invocations of *ar* and *ranlib*:

```
$ export IPCMD_SEMID=$(ipcmd semget) # create a semaphore
$ ipcmd semctl setall 1              # initialize it to 1
$ ./configure AR='ipcmd semop -u -1 : ar' \
              RANLIB='ipcmd semop -u -1 : ranlib'
$ make -j 8
...
$ ipcrm -s $IPCMD_SEMID              # remove semaphore
```

make


mpich2-1.4 build time
avg. of 3 runs
seconds
160
140
120
100
80
60
40
20
0
make -j 8
make

bzip2

- bzip2 is a standard compression utility
- pbzip2 is a parallel (multi-threaded) implementation of bzip2
- bunzip2 can decompress a file that is the concatenation of multiple compressed files
- Using ipcmd, a simple framework that allows partitioning of input and execution of (serial) bzip2 on each partition was created (parallelpipe.sh)

bzip2

Syntax:
parallelpipe.sh [-p PROCS] [-n SLOTS] <partitioner> <args...> : <filter> <args...>

-p PROCS	Maximum number of filter processes to concurrently execute.
-n SLOTS	Maximum number of partitions to buffer.
partitioner	User-specified program to partition the standard input of parallelpipe.sh. One instance of this program will be spawned.
filter	User-specified program. For each input partition, a separate instance of filter will be spawned; it will read the partition on its standard input. The standard output of each filter instance will be concatenated and written to the standard output of parallepipe.sh

bzip2

The bzip2 program was used as the filter, while the following shell script was used as the partitioner:

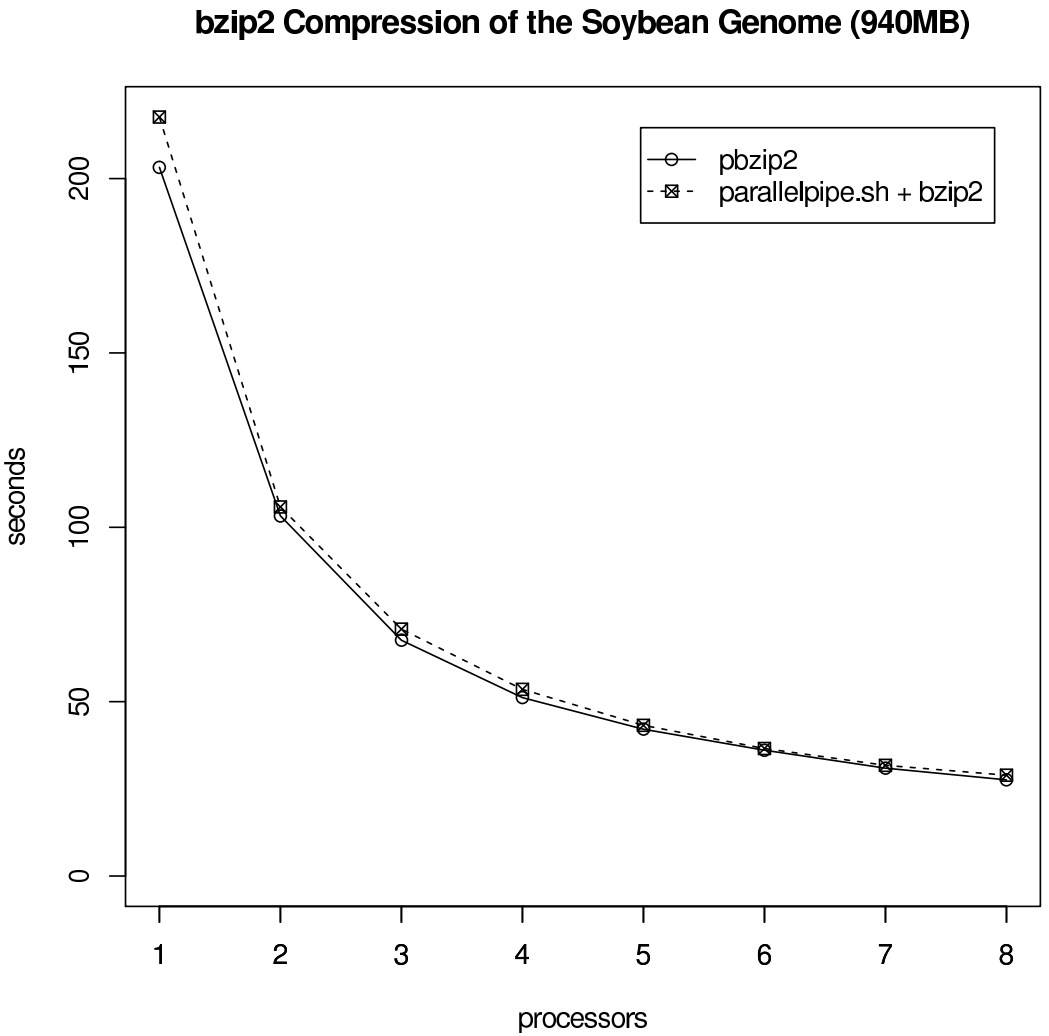
Listing 1: bzip2 partitioner

```
#!/bin/sh

readonly PARTITION_SLOT_SEM=0 WRITE_SEM=1 READ_SEM=2

while true
do
  ipcmd semop $PARTITION_SLOT_SEM=-1 $WRITE_SEM=-1
  dd of=$PARTITION_PATH bs=2048k count=1 2> /dev/null
  if [ ! -s $PARTITION_PATH ] # end of input
  then
    break
  fi
  ipcmd semop $READ_SEM=+1 # signal partitioner coordinator
done
```

bzip2


bzip2 Compression of the Soybean Genome (940MB)
pbzip2
parallelpipe.sh + bzip2
seconds
0
50
100
150
200
1
2
3
4
5
6
7
8
processors

Bugs Reported During ipcmd Development

POSIX XSI IPC specification:

- http://austingroupbugs.net/view.php?id=329
- http://austingroupbugs.net/view.php?id=335
- http://austingroupbugs.net/view.php?id=366
- http://austingroupbugs.net/view.php?id=502
- http://austingroupbugs.net/view.php?id=532

OpenBSD implementation of semctl():

- http://www.mail-archive.com/bugs@openbsd.org/msg00958.html

OS X bugs: one in the implementation of semctl(), the other in the implementation of semop()

Questions

Questions?