Basic Multiprocessing in UNIX

With Examples

Parallel Applications

- Modern computers have multiple CPU cores (and/or multiple CPUs) on board
- We have to be able to utilize the computing power by parallelizing our tasks

CPU Information

- Linux computer: /proc/cpuinfo
- Cat /proc/cpuinfo example:

processor : 0
vendor_id : AuthenticAMD
cpu family : 15
model : 65
model name : Dual-Core AMD Opteron(tm) Processor 8220
stepping : 3
cpu MHz : 2800.000
cache size : 1024 KB
physical id : 0
siblings : 2
core id : 0

cpu cores : 2
fpu : yes
fpu_exception : yes
cpuid level : 1
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt rdtscp lm 3dnowext 3dnow pni cx16 lahf
_lm cmp_legacy svm extapic cr8_legacy
bogomips : 5625.16
TLB size : 1024 4K pages
clflush size : 64
cache_alignment : 64
address sizes : 40 bits physical, 48 bits virtual
power management: ts fid vid ttp tm stc

Processes in UNIX

- UNIX is natively parallel operating system
- A *process* is an instance of running a program
- Each process has a unique *process id*
- Shell command “ps” gives the list of all running processes

Using the shell commands

- In any UNIX shell, “&” will run the command in background.
- The command will run in its own shell, which is a child of the current shell

```
[alekseyz@genome10]$ run_command.sh &
```

- “wait” command will wait for all child processes in the current shell to finish

Example of & and wait

- In bash:

```
#!/bin/bash
let NUM_CPUS=`cat /proc/cpuinfo |grep processor|tail -1|awk '{print $NF+1}'`
let counter=1;
let cpu_counter=1;
echo "Total processes to run:"$max_counter
echo "Simultaneously running:"$NUM_CPUS

while [[ $counter -le $1 ]];do
   while [[ $cpu_counter -le $NUM_CPUS &&  $counter -le $1 ]];do
        ./echo_sleep_echo.sh &
        let counter=$counter+1
        let cpu_counter=$cpu_counter+1;
    done
    let cpu_counter=1;
    wait
done
```

```
#!/bin/bash
echo "Sleeping 10 seconds in shell "$$
sleep 10
echo "Done"
```

Using fork() and wait() in C

- The fork() system call is the basic way to create a new process. fork() is used to produce child shell.
- Returns twice(!!!!)
- fork() causes the current process to be split into two processes - a parent process, and a child process.
- All of the memory pages used by the original process get duplicated during the fork() call, so both parent and child process see the exact same memory image.

fork() continued

- When fork() returns in the parent process, its return value is the process ID (PID) of the child process.
- When it returns inside the child process, its return value is '0'.
- If for some reason fork() failed (not enough memory, too many processes, etc.), no new process is created, and the return value of the call is '-1'.
- Both child process and parent process continue from the same place in the code where the fork() call was used.

Child processes

- When a child process exits, it sends a signal to its parent process, which needs to acknowledge it's child's death. During this time the child process is in a state called *zombie*.
- When a process exits, if it had any children, they become *orphans*. An orphan process is automatically inherited by the *init* process, and becomes a child of this *init* process.
- When the parent process is not properly coded, the child remains in the zombie state forever. Such processes can be noticed by running the “ps” command, and seeing processes having the string "<defunct>" as their command name.

Simple fork() and wait() example

```
#include <stdio.h>
#include <unistd.h>     /* defines fork(), and pid_t.      */
#include <sys/wait.h>   /* defines the wait() system call. */
int main(){
pid_t child_pid;
int child_status;

child_pid = fork();
  switch (child_pid) {
    case -1:
        perror("fork");
        exit(1);
    case 0:
        printf("I am the child, Hello world\n");
        sleep(10);
        exit(0);
    default:
        printf("I am the parent, waiting for the child process %d to exit...
   \n",child_pid);
        wait(&child_status);
        printf("I am the parent, child process %d exited with status
   %d\n",child_pid,child_status);
  }
}
```

InterProcess communication

- One can prescribe what each child does in the fork() call
- It is helpful if parent could communicate with child (e.g. report progress, get data)
- Easiest way for parent and child to communicate is through *pipe*

Communication

- One can prescribe what each child does in the fork() call
- It is helpful if parent could communicate with child (e.g. report progress, get data)
- Easiest way for parent and child to communicate is through *pipe*

Using pipes

- *Anonymous pipe:* A pipe is a one-way mechanism that allows two related processes (i.e. one is an ancestor of the other) to send a byte stream from one of them to the other one.
- The order in which data is written to the pipe, is the same order as that in which data is read from the pipe.
- The system assures that data won't get lost in the middle, unless one of the processes (the sender or the receiver) exits prematurely.

pipe()

- The pipe() system call is used to create a read-write pipe.

- pipe() takes as an argument an array of 2 integers that will be used to save the two file descriptors used to access the pipe. The first to read from the pipe, and the second to write to the pipe.

Using pipe()

```
/* first, define an array to store the two file
   descriptors */
int pipes[2];


/* now, create the pipe */
int rc = pipe(pipes);

if (rc == -1)
{
/* pipe() failed */
perror("pipe");
exit(1);
}
```

pipe() example -- main

```
int main()
{
int data_pipe[2]; /* an array to store the file descriptors of
   the pipe. */
int pid;
int rc;

rc = pipe(data_pipe);
if (rc == -1) { perror("pipe"); exit(1); }

pid = fork();
switch (pid)
   {
   case -1:
        perror("fork"); exit(1);
   case 0:
        do_child(data_pipe);
   default:
        do_parent(data_pipe);
   }
}
```

pipe() example -- parent

```
void do_parent(int data_pipe[]) {

int c; /* data received from the user. */
int rc;

/* first, close the un-needed read-part of the pipe. */
close(data_pipe[0]);

while ((c = getchar()) > 0)
   {
        rc = write(data_pipe[1], &c, 1);
        if (rc == -1)
        {
        perror("Parent:write");close(data_pipe[1]);exit(1);
        }
   }
close(data_pipe[1]);
exit(0);
}
```

pipe() example -- child

```
void do_child(int data_pipe[]) {
int c; /* data received from the parent. */
int rc;

/* first, close the un-needed write-part of the pipe. */
close(data_pipe[1]);

while ((rc = read(data_pipe[0], &c, 1)) > 0)
   {
        putchar(c);
   }
exit(0);
}
```

Acknowledgements

- Some examples were taken from:

  http://users.actcom.co.il/~choo/lupg/tutorials/multi-process/multi-process.html#process_creation_fork_syscall