Operating Systems Class - Assignment 1

João Ferreira & Pedro Cristina Marques
Department of Informatics Engineering
University of Coimbra
jpbat@student.dei.uc.pt | pgcm@student.dei.uc.pt
2009113274 | 2007184032

September 2011

Introdução

Neste projecto foi-nos proposto calcular uma aproximação do número irracional π, através do método de `Monte Carlo`, tirando partido do princípio da programação concorrente.

O números de pontos usados no teste é definido como uma constante global (`resolution`) no ficheiro `monteCarlo.c`. É utilizada também uma variável booleana (`debugMode`) que nos permite executar o programa em modo de `debug`. Além disto é utilizada ainda outra constante (`processes`) que indica o número de processos a ser usado.

Processo Pai

O processo pai começa por bloquear todos os sinais com a excepção do `SIGUSR1` que é tratado com o `master_sigusr1_handler()` e do `SIGINT` que por sua vez é tratado com o `master_sigint_handler()`. De seguida vai criar uma matriz de `pipes` para garantir comunicação com todos os filhos. Após isto cria o número de processos definido anteriormente, e cada um destes por sua vez chamam a rotina `worker()` recebendo do processo pai, o `id` correspondente a si, o número de pontos que devem calcular, o número total de processos, a tabela de `File Descriptors` e uma boolean a informar se o programa está ou não a correr em `debugMode`. O número de pontos que cada processo deve calcular é dado através da fórmula: $\frac{TotalDePontos}{NumeroDeProcessos}$.

Após a criação de cada filho o processo pai fecha a ponta de escrita do `pipe` correspondente ao processo que acabou de ser criado, uma vez que esta é desnecessária na nossa abordagem ao problema. Seguidamente o processo pai faz uso da função `select()` para receber as mensagens enviadas pelos processos filhos, de referir que segundo a nossa implementação é feita a cada leitura uma verificação do número de bytes lidos. Após ler as mensagens de todos os processos filhos, o processo pai envia um `SIGUSR1` através da rotina `kill()` e faz `wait(NULL)` a cada processo filho, para que estes terminem a sua execução, evitando assim a existência de processos `Zombie`. Por fim o processo pai faz o calculo do π e imprime-o, desbloqueando depois todos os sinais.

Podemos referir ainda três aspectos: a rotina `master_sigusr1_handler()` apenas informa que o programa recebeu um `SIGUSR1`, enquanto por outro lado a rotina `master_sigint_handler()` informa que recebeu um `SIGINT`, tendo o cuidado de matar os filhos e fazer `clean shutdown`. Por fim de

referir que o `debugMode` é apenas uma série de prints que podem ser feitos para procurar eventuais precalços durante a execução do programa.

Rotina Worker()

A rotina `worker()` começa por bloquear todos os sinais, com a excepção do `SIGUSR1`, que trata com a rotina `son_sigusr1_handler()`. Seguidamente fecha todas as pontas dos `pipes` que não lhe dizem respeito ou que não vai utilizar. Após isto gera uma nova `seed` para o cálculo de números aleatórios, uma vez que caso esta fosse criada no processo pai, todos os filhos iriam gerar o mesmo conjunto de pontos. Após isto conta o número de pontos que estão circunscritos ao quarto de circunferência de raio 1, e envia esse número através do uso de um `pipe` para o processo pai.

O processo faz uso da rotina `pause()`, ficando bloqueada à espera de um sinal que lhe vai ser enviado pelo processo pai. Após receber este sinal, o processo imprime no display uma mensagem de confirmação, desbloqueia todos os sinais, fecha a ponta do pipe que usou e morre.

SpeedUp & Occupancy

Para termos noção de qual era o nosso `speedup` e a `ocupância` do nosso `CPU`, foi desenvolvido um script em `bash` que executa cada uma das versões (sequencial e concorrente) 30 vezes, guardando cada um deles para ficheiro. Após isso chama outro script que calcula a média, e o desvio padrão de cada uma das versões. Assim chegamos aos seguintes resultados:

Points	Processes	Avg(ms)	StdDev(ms)	SpeedUp	Occupancy
100M	1	3882	40	———	———
100M	2	1951	79	1.99	99.49%
1000M	1	39620	1402	———	———
1000M	2	19423	1142	2.04	101.99%

Para analisar a tabela convém primeiro referir as seguintes fórmulas: `SpeedUp` $= \frac{AvgSequencial}{AvgConcorrente}$ e `Occupancy` $= \frac{SpeedUp}{NumeroDeCores}$ e também que todos os testes foram corridos numa máquina com 2 `cores`, sendo que cada um deles tem `2.80Ghz` de velocidade.

Conclusão

Assim podemos concluir que na versão em que o programa é executado com cem milhões de pontos, a `Occupancy` é bastante boa, tal como o `SpeedUp`, visto que este é bastante próximo de 2, ou seja o limite virtual do mesmo, já que esse é o número de cores do computador de teste. Já na versão de mil milhões de pontos denota-se que o `SpeedUp` é ligeiramente superior a 2, o que acontece porque mesmo apesar de correrrmos 30 testes, provavelmente ao executar a versão sequêncial o processador poderia estar a correr um outro processo que estivesse a consumir recursos, aumentando assim o tempo de execução do algoritmo. A `Occupancy` é também ligeiramente superior a 100%, o que é normal, uma vez que o valor do `SpeedUp` é usado no cálculo da mesma.