CS4202 Computer Architecture - Process Scheduling for Heterogeneous Systems

140011146

November 19, 2017

1 Introduction

CPU scheduling is vital for the multitasking environment of modern operating systems. Processes waiting for I/O can be switched out to prevent wasted CPU cycles, but any process can be pre-empted for better user interactivity, efficiency and fairness [8]. As a result, it is important to understand the effects of scheduling on performance.

First, a round-robin scheduling approach for CPUs is looked into, discussing how the approach works and where the approach performs well or poorly.

Next, a genetic algorithm was implemented to explore the search space of possible schedules with the GEM5 simulator on two benchmarks. The schedules produced by the genetic algorithm are then compared with both the default approach of the simulator and a completely random schedule to gain an understanding of the scheduling optimisation space.

2 Task 1 - Round-robin scheduling

The round-robin CPU scheduling approach is similar to a simple First-Come First-Served scheduling approach, but all processes bursts are limited by a time quantum. This ensures fairness, giving all processes a chance to run, but comes at costs such as high waiting time and high turnaround time. There is much research going into optimising a round-robin scheduler, usually by means of calculating an optimal time quantum dynamically [4] [5] [6] or introducing some form of priority [7].

In the simple round-robin approach, a circular queue of processes that are ready to be executed is maintained by the scheduler. New processes join at the end of the queue. The scheduler chooses and removes the first process in the queue to run [8]. Any running process will be pre-empted if they run longer than the time quantum. Processes that make block or finish before the time quantum

are also moved off the CPU [9]. A very long time quantum will effectively be the same as a FCFS approach, resulting in poor response and waiting time if there are processes with long bursts. On the other hand, a very short time quantum results in a large computational overhead of many context switches between the processes, wasting time and CPU cycles on context switching [1] rather than useful computation. Therefore, the length of the time quantum heavily impacts the performance of the round-robin approach. For example, consider the following processes in the ready queue.

Process	**Burst time**
P_1	17
P_2	3
P_3	4

A time quantum of 3 would result in the following schedule and wait times:

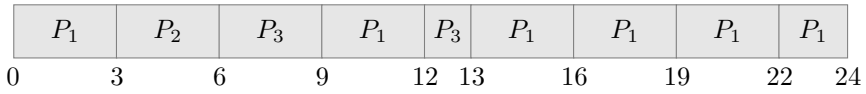


Figure 1: Schedule with a time quantum of 3

Wait times:

- P_1: 7 time units
- P_2: 3 time units
- P_3: 9 time units

Average wait time = $19/3 = 6.33$ time units.

However, a time quantum of 10 would result in a much longer average wait time.

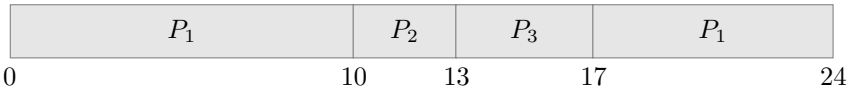


Figure 2: Schedule with a time quantum of 10

Wait times:

- P_1: 7 time units
- P_2: 10 time units
- P_3: 13 time units

Average wait time = $30/3 = 10$ time units.

It can be seen from the example how the choice of time quantum affects the waiting time, response time and number of context switches required. Round-robin is strong in cases where all processes bursts have similar lengths, or fairness is of greater importance to the system. For example in load balancing for parallel computation or time-sharing

Changing the size of the time quantum affected the average waiting time, response time and number of context switches needed. as process 2 and 3 had to wait much longer before getting a chance to run. However, this comes at a trade-off of less context switches. With a quantum of 10, only 3 context switches were required, whereas with a quantum of 3, 8 context switches were required.

An issue with the simple round-robin scheduling approach is the long average waiting time. Given n processes in the ready queue with a time quantum of t, each process only gets t time to run every nt time units and must wait at most $(n-1) \times t$ before each turn.

Round-robin performs well for load balancing if the processes all have similar bursts times.

Some processes want/should have more share of the processor than others (concept for priority)

Long queue leads to high response time and high turnaround time as a simple round-robin approach adds new processes to the tail of the queue, giving a response time of at most $(n-1) \times t$ time units where n is the number of processes in the queue and t is the time quantum.

3 Task 2 - Search for high performance schedules

3.1 Methodology

A genetic algorithm was chosen as a means to explore the search space of binary schedules. Because of the large optimisation space of schedules, a genetic algorithm with mutation adds elements of randomness to help explore more of the search space compared to a hill climbing algorithm which can get easily stuck in local maxima. Furthermore, hill climbing with random restart will only try to explore another unrelated area of the search space for a maximum without taking into account the previous local maxima. With a genetic algorithm, the next generation is created based on the previous generation, so good schedules can be maintained and modified through each generation. The fitness function used to evaluated a schedule is the total time taken of the execution of that schedule from the output of the simulation. The less time taken, the fitter that

individual. As a schedule is a binary string, this maps perfectly to a genetic chromosome, where each gene is a single bit.

The genetic algorithm that was implemented follows three basic steps [2]:

1. **Selection** - Select parents to use for reproduction of next generation. The fitter the chromosome, the more likely it will be chosen for reproduction. **Elitism** is also implemented, keeping exact clones of the most fit individuals so the best schedules are not lost.

2. **Crossover** - Mix the genes of the parents to create two new child schedules.

3. **Mutation** - Genes (bits) of all child schedules have a chance to be mutated to increase diversity and prevent the algorithm from being stuck in a local maxima.

3.1.1 Selection

A tournament selection method was used to select parents for reproduction. This method randomly selects a few chromosomes from the population, and chooses the most fit individual as the "winner" of the tournament. Tournament selection was used for its simplicity and flexibility. The tournament size can be decreased to allow weaker individuals more chance to be selected for more diversity or increased to give them less chance for faster convergence.

The same individuals are allowed to be selected multiple times for reproduction for better convergence. It is also done so the randomness in the crossover and mutation stages do not lead to a complete loss of strong genes. Additionally, a small random number of elites are selected to be clone in each generation as part of **Elitism** to keep the fittest individuals. This is done to ensure the best schedules survive every generation as the effect of crossover and mutation on creating consistently better schedules is not known due to the size and complexity of the search space. The number of elites kept is randomised so the same elites are not kept every generation if the algorithm is at a local maximum.

3.1.2 Crossover

The simulator is deterministic, that is to say the same schedule will always take the same time to run. Additionally, if two schedules share the same initial n bits, they will run identically until they differ. This means that where the bits change in the binary stream matters. Given a binary schedule, if the first few bits are modified, the rest of the schedule may be interpreted completely differently even if no other bits are modified. If only the last few bits are modified, then the schedule is only different at those last few bits. The crossover step for genetic algorithms is quite suited for this behaviour. The children created by the two parents will each contain the first n bits from each parent and the rest from

the other parent. If both parents were good schedules, this explores changing and mixing the second half of the schedule to potentially find an even better schedule without losing the good performance from the first half of the schedule.

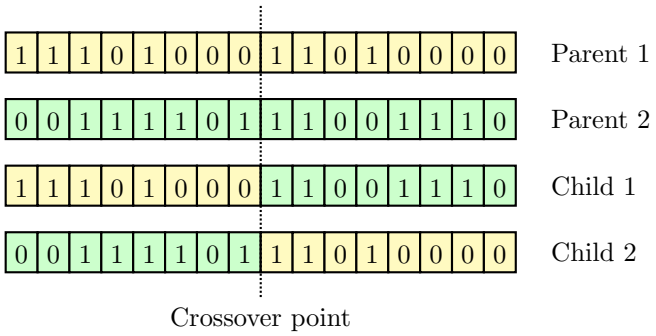


Figure 3: Example reproduction of children

There is only one single crossover point in the implementation as more complicated crossover techniques would take less advantage of the deterministic behaviour of the simulator and schedule. The crossover point is chosen randomly for more diversity as the same individual can be chosen as a parent multiple times.

3.1.3 Mutation

The mutation aspect of the algorithm will add diversity to the population, potentially changing a child schedule completely. This leads to a loss of a potentially good schedule, especially if the mutation occurred early and the parents were fit, but allows searching through more of the optimisation space to try and find even better schedules.

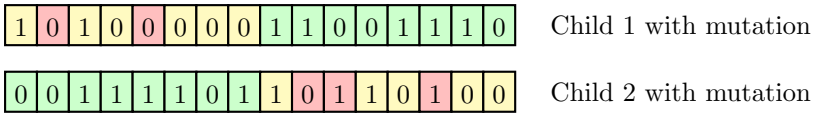


Figure 4: Children from figure 3 with mutation

It is important to note that although the combination of selection, crossover and mutation can lead to a systematic way of searching for better schedules while taking into account previous strong genes and keeping the best individuals, the complexity of the search space and lack of understanding on how changes to the bits directly affect the time taken make the performance of genetic algorithm unpredictable. It could well be that randomly creating schedules is an easier and faster method to find the fastest schedule.

3.1.4 Parameters and other optimisations

Optimise size of schedule as much as possible to prevent entire lower half being unused

- because we don't know how schedule affects time, unsure if crossover/selection/mutation strategies are suitable

- if schedule is too short, then the simulator will fail and return 0 for that fitness -¿ this allows removing potential bad schedules

3.2 Experimentation

The two benchmarks `sched-blackscholes_gen` and `sched-bodytrack_gen` are used in the experimentation. Initially, a naïve experiment was run using arbitrary parameters to see how well the genetic algorithm performed after many generations. Afterwards, the experiment was improved by adjusting the parameters to improve diversity. The results of both experiments are later compared with the default weighted RNG scheduler and a completely random approach.

3.2.1 Experiment 1

In the first naïve experiment, the following parameters were chosen:

- **Population size**: 25
- **Chromosome length**: 100000
- **Mutation probability per gene**: $\frac{1}{(l/10)}$ where l is the length of the chromosome
- **Crossover point**: Random
- **Selection**: Tournament with 3 competitors

The population size was chosen as a size that was not too small or too large to start with. Fine tuning the population size for the specific task of finding a fast schedule will require more work, but starting with a medium sized population will give an indication as to whether this is too small or too large. A population that is too small will be stuck in a local maximum and unable to find better solutions, while a population that is too large would lead to a slow rate of convergence [3].

3.2.2 Experiment 2

From the first experiment, it was noticed that the algorithm starts to stagnate quickly. After a few generations, either because of a lack of mutation or population size, a large portion of each generation end up with the same schedule and fitness with little diversity. To fix this issue, the population and mutation

probability is increased to increase diversity. The larger population and higher mutation rate explores more of the search space, but leads to slower convergence. To be able to keep strong individuals with the increased population and mutation, the number of individuals used in the tournament selection is also increased.

Increased mutation also due to local maxima. Perhaps crossover is not effective because if the schedule is good, only some portion of the top half of the binary is used, so low mutation rate where mutation happens in lower half has no effect.

- **Population size**: 40
- **Chromosome length**: 100000
- **Mutation probability per gene**: $\frac{1}{(l/100)}$
- **Crossover point**: Random
- **Selection**: Tournament with 5 competitors

3.3 Results

3.3.1 Benchmark properties

The two benchmarks chosen for this experiment are `blackscholes` and `bodytrack`. First, a quick overview on how the performance of the default and random approach on the benchmarks. This will show how well these approaches work as well as if there are any distinct differences between the two benchmarks. Both the random and default approaches were run for 250 simulations.

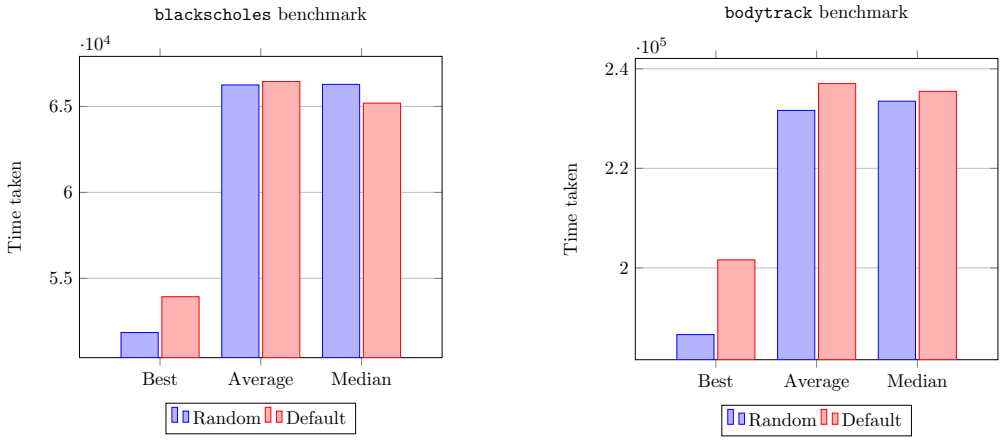


Figure 5: Best and average times taken by the random and default approach.

There are a few interesting points to note about the random and default approaches. First, the random approach is able to find a faster schedule than the default. This is interesting as one would expect the default approach to be able to come up with good and fast schedules. This shows that better schedules can be found through search that improve upon those made by the default scheduler. Following this, it can be seen that the average times for random and default are quite similar. The median was also taken as an extra check to make sure the average was not skewed by anomalies, such as when the time taken is set to 0 for simulations which failed. This shows how the two approaches are similar in the kind of schedules being produced and an algorithm that can improve on a random approach should also improve against the default approach.

Another statistic of note is the variance, or standard deviation of the random and default approach. Here the standard deviation is divided by the mean to get the variability relative to the mean and to be able to compare the two benchmarks.

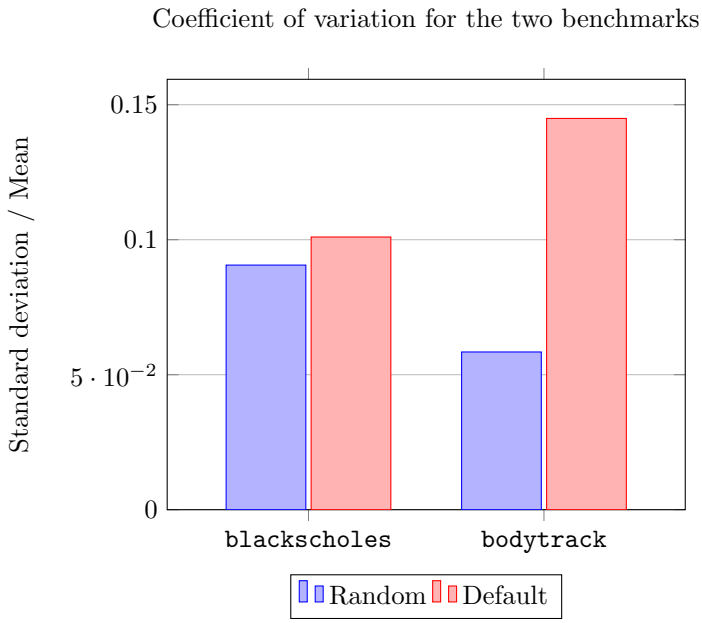


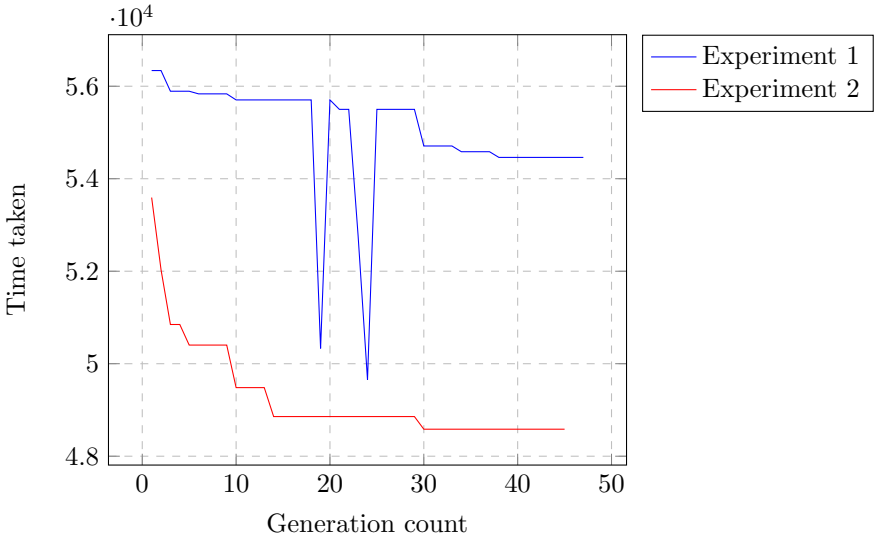
Figure 6: Coefficient of variation by the random and default approach.

Again, it can be seen that the variation in schedules for the default approach is not necessarily better than that of random schedules. This may be due to some mistake or assumption by the scheduler which led to slower schedules than expected. A certain approach by the scheduler may be very effective for other benchmarks, but less useful for these benchmarks. Additionally, in the `bodytrack` benchmark, the low variation for random yet high variation for the

default suggests the benchmark may be less sensitive to changes in the schedule. This could be due to the order of processes having little effect on the overall time taken.

3.3.2 Genetic algorithm results

Genetic algorithm experiments on the `blackscholes` benchmark



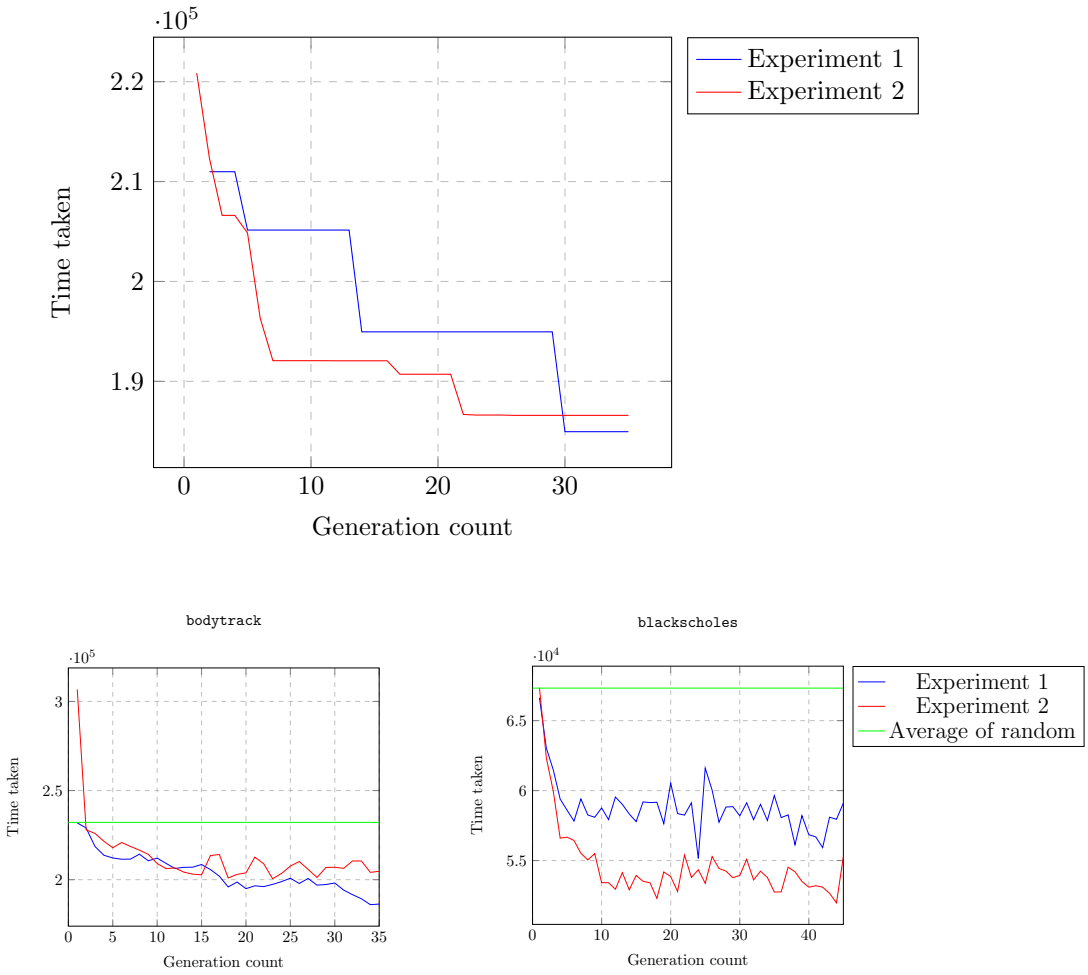


Figure 7: Mean time taken in each generation by experiments

- more time on genetic algorithm should tend to better result -¿ especially if given more diversity - random gets faster schedule quicker, but less likely over time to find good schedule, whereas genetic algorithm guarantees good schedule over time as long as high diversity. - ga can never/rarely get optimal schedule

3.4 Evaluation

References

[1] *Context Switch Definition*. May 2006. URL: http://www.linfo.org/context_switch.html.

[2] *Introduction to Genetic Algorithms*. 1996. URL: https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/hmw/article1.html.

[3] V.K. Koumousis and C.P. Katsaras. "A saw-tooth genetic algorithm combining the effects of variable population size and reinitialization to enhance performance". In: *IEEE Transactions on Evolutionary Computation* Volume 10, Issue 1 (Feb. 2006).

[4] Rami Matarneh. "Self-Adjustment Time Quantum in Round Robin Algorithm Depending on Burst Time of the Now Running Processes". In: 6 (Oct. 2009).

[5] Manish Kumar Mishra and Dr. Faizur Rashid. "An Improved Round Robin CPU Scheduling Algorithm with Varying Time Quantum". In: *International Journal of Computer Science and Engineering* Vol 4, No. 4 (2014). URL: http://airccse.org/journal/ijcsea/papers/4414ijcsea01.pdf.

[6] Abbas Noon, Ali Kalakech, and Seifedine Kadry. "A New Round Robin Based Scheduling Algorithm for Operating Systems: Dynamic Quantum Using the Mean Average". In: *International Journal of Computer Science and Engineering* Vol 8, Issue 3, No. 1 (2011). URL: https://arxiv.org/pdf/1111.5348.pdf.

[7] Ishwari Singh Rajput and Deepa Gupta. "A Priority based Round Robin CPU Scheduling Algorithm for Real Time Systems". In: *International Journal of Innovations in Engineering and Technology* Vol 1, Issue 3 (2012). URL: http://ijiet.com/wp-content/uploads/2012/11/1.pdf.

[8] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts Essentials*. 2nd. Wiley Publishing, 2013. ISBN: 9781118804926.

[9] Ajit Singh, Priyanka Goyal, and Sahil Batra. "An Optimized Round Robin Scheduling Algorithm for CPU Scheduling". In: *International Journal of Computer Science and Engineering* Vol. 02, No. 07, (2010). URL: http://www.enggjournals.com/ijcse/doc/IJCSE10-02-07-83.pdf.