Resource Allocation by Economic-Based Methods

Michael Nahas

1.0 Introduction

Resource allocation is a major function of an operating system, and it is also the primary function of an economy. This research examines if economic-based methods for resource allocation are applicable to three items in the OS; the CPU, the disk, and the memory.

So, why use economic models for resource allocation in computers?

The computer is a very specialized environment, but money systems have been flexible enough to survive in all kinds of environments. One benefit of using a money system is that they generally result in fair and efficient allocation. Another benefit is that each entity can make its own decision of how much each resource is worth to it. Lastly, money systems have advantages in distributed systems, providing a good way to compare the demand for each resource on a node or between nodes.

Money systems result in fair allocation. This is useful in an OS, where resources have to be divided on a process by process basis and on a user by user basis. A good example of how to use this in an OS is by giving each user a fixed amount of income, which he or she can divide among all of his or her programs running on a system. This prevents a user from dominating all the resources by starting multiple programs. Money systems also achieve efficient allocations, because the price of scarce resources goes up, applications are encouraged to find less expensive solutions.

Another potential benefit of an economic-based resource manager is the ability for programs to decide how much to bid for each resource. In this research, we had each program's bid determined by the OS, but that bid was based on a few pieces of constant data about each task, e.g. its working set size. It is certainly possible to have each program make is own decision of what to bid for each resource. This kind of customization would require more work from the programmer, but would result in better performance.

Lastly, money systems have a great advantage in distributed systems, since prices for a resource on one node can be directly compared with prices on another node. It is very useful for deciding where a task should be started, or when to move a task to a different node. This kind of comparison is difficult to do with conventional resource allocators.

2.0 Related Works

Research into economic models for resource allocation has been going on since before 1968, when a paper was published in the Communications of the ACM involving bidding for a CPU. Only, in this case, the CPU was a PDP-1 at Harvard University and the bidding was done by humans, because the compute time was too valuable.

The Harvard system involved dispensing imaginary "yen" to each individual in the department, and having each person bid in yen for the PDP-1. The system had a few rules to keep things friendly, such as when someone outbids someone for only part of the time they had reserved, the original bidder must be left with a continuous chunk of time. Once the individual is done using his compute time, the yen bid for it return to his or her account. As a matter of convenience, bidding would stop 24 hours before the time became available, to prevent people from being outbid right before they sat down to work.

Harvard appeared to be happy with its system for allocating time. User priority could be easily controlled by increasing or decreasing their amount of yen. The computer ended up being used almost around the clock. It effectively distributed time so that hotly contested times had short blocks and less desirable times had long, cheap blocks of time. Another conclusion was that although users disliked being able to be preempted by higher bids, they complained less when their time was preempted by maintenance.

Another excellent look at economics in computing is the research done at IBM Watson and Columbia by Ferguson, Yemini, and Nikolaou. The paper looked at using bidding for CPU and communication links in a distributed computer to perform load balancing.

They simulated a 9 node machine with mesh interconnect, with jobs arriving equally at each node in a Poisson distribution. Each job's responsibility was to execute and get its result data back to the node where it was created, and spend as little of its fixed amount of money as possible. Each CPU tried to maximize its own income, and advertise its price to its local neighbors in the mesh. Each interconnect tried to maximize its own income.

The two algorithms they used for bidding seemed haphazard and did not show any knowledge of game theory. In fact, they reported that giving each of these variable compute time process a constant amount of money or a random amount of money made the system perform just as well as when the processes were given money proportional to their compute time. This was a result of their bidding scheme, where the process willing to spend the most on the resource could easily lose the auction.

Its handicapped bidding scheme may have prevented the best task from moving, but on average, it did find a good task to move and balanced the load well. It performed much better than a system without load balancing, and performed slightly better than HOP 1, a comparable algorithm for mesh network load balancing.

I would also like to mention that the system of bidding for the CPU presented in this paper has been done before. I found a paper on the web describing it exactly, but that website has been reorganized and I was unable to find any reference to the paper. I did not find anything similar to my system of bidding for the disk or memory.

3.0 The Simulation Environment

The simulated environment is a simple stand-alone computer. The resources allocated by the OS are a single CPU, a single disk, and K physical pages of memory for use by the tasks. Competing for these resources is a pool of UNIX-style processes that have a single kernel-level thread which blocks on an I/O request or on a memory fault. This section describes the behavior of both the resources and the processes.

The simulator is a time-step simulator. During each time step, the CPU can execute a process in the CPU queue and the disk can process, or continue processing, a process in the I/O queue. The simulation runs for a fixed number of time steps, and then outputs the state of the processes.

The winner out of the CPU queue always gets the CPU for a fixed size quanta. For comparison to disk speeds, this quanta length was considered to be 1 millisecond. The process that gets the CPU ends up doing one of 3 actions: faulting, requesting I/O, or finishing the quanta. Because a time-step simulator was used, a process that faulted or requested I/O was recorded as having completed 0% of the quanta; an event driven simulator might have allowed processes to complete fractions of a quanta.

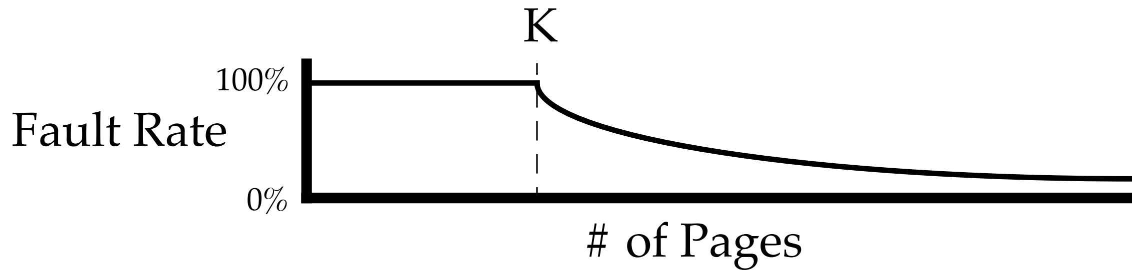
The winner out the I/O queue gets the disk for a variable length of time, depending on the location of the disk head and the block being read. The disk head can be on one of 20 “tracks” and it takes one time step to move one track, giving the disk a maximum seek time of 19 milliseconds. (Moving from track 0 to 19 only involves 19 moves, not 20.) Which gives our simulated disk an average seek time of 5.7 milliseconds. Rotational latency is a random value between 0 and 5 time quanta, simulating a rotational speed of 10,000 rpm. Lastly, the disk cannot be preempted, so once a process has been allocated the disk, the I/O request will complete.

A task which has faulted needs to complete 2 I/O requests, one to write out the old page, and one to read in the new one. Obviously, this approach does not account for code pages, which do not need to be written out to disk, and newly allocated pages, which do not need to be read in from disk. It was considered to costly in terms of time to have included that functionality.

The last resource is memory pages. Initially, all the pages belong to the null process and demand paging allocates them to the user processes. When a fault occurs, the allocator selects a physical page as a destination for the new virtual page. Ownership of that page is transferred to the faulting process and it is marked as invalid. After the old page has been written out, and the new one is read in, the page is marked valid and the process is unblocked.

When does a process do I/O or encounter a fault?

The chance of a fault occurring is determined by the number of valid pages owned by a process, and two numbers used by the simulator: the working set size and the sloping factor, a number between 1.0 and 0.0. If the number of pages owned by the process is less than the working set size, the fault rate is 1.0. If the number of pages owned by the process is greater than or equal to the working set size, the fault rate is the sloping factor raised to the number of pages beyond the working set. The graph of fault rate vs. the number of pages owned by the process looks something like this:



The chance of a process doing I/O during a quanta is always a fixed percentage. The I/O is always one disk block, and does not require allocating memory. Which block to access is chosen randomly.

4.0 The Control: A Conventional Scheduler

For the conventional scheduler, I chose Start-time Fair Queuing for the CPU, SCAN for the disk, and a form of global LRU for the memory.

Start-time Fair Queuing (SFQ) is one of many fair sharing algorithms which try to give each active task a portion of the CPU that is proportional to its priority. I chose to use this algorithm because I expected the economic-based scheduler to provide a relatively fair sharing of resources and wanted to compare it to a fair sharing algorithm.

SCAN, better known as the elevator algorithm, is a common disk scheduling algorithm. The disk head keeps moving in one direction until there are no more requests in that direction. Its benefits include that it is nearly impossible to starve a requesting task.

LRU is the ideal paging algorithm, but my model of tasks says nothing about with pages are touched during a quanta, only their probabilities of being touched. I attempted to simulate this by generating a random number between 0 and 1 at each CPU quanta, and comparing it to the fault rate of each page. If that number is less than the fault rate for the Nth page, then the N page was considered touched, and the clock for that page is updated. The globally least recently used page is selected as the location for the new page in memory.

5.0 The Experiment: An Economic-Based Scheduler

The economic-based scheduler is based on game theory bidding. Each process has a balance and receives an income each quanta. With that money, the process can bid on CPU, I/O and physical pages in memory. The bidding is done by a game theoretic model of bidding, which will be explained before the individual schedulers are explained.

5.1 Game Theory

Let's first look at an auction for 1 item. Lets assume there are any number of people bidding for the item, and each as a unique amount at which they are willing to pay for the item. We can order the people by the amount they value the item and call them **A**, **B**, **C**, ... Now, **A** values the item more than any other bidder, and **B** values the item more than any other bidder except **A**. In order to win the item, **A** must make a bid higher than anyone else is willing to pay, but it is to **A**'s advantage to not waste his money. So the ideal bid for **A** is what **B** is willing to pay for it, plus some small value, which in theory we can say is insignificantly small. So **A** wins the item at the price that **B** is willing to pay.

Let's now take a look at an auction for N items, where a bidder can only win at most 1 of the items. We can do our ordering again, and end up with **A**, **B**, **C**, ... Now the N people at the head of that list only have to out bid the N+1 person in the list. So the N people win at the price that the N+1 person is willing to pay.

Now, lets expand the auction so that a bidder can win more than one item. Lets assume **A** is willing to pay P_1 for one item, P_2 each for 2 items, P_3 each for 3 items, etc. We can reduce this problem to the previous one by having **A** send N bidders as proxies to an auction where each bidder can only win one item. Now, the first proxy bidder is willing to bid P_1 for the item, and second is willing to bid P_2, the third P_3, etc. Now, if K of **A**'s proxy bidders win at the auction, **A** has won K items at a price lower than P_K for each item.

5.2 Bidding for the CPU

As stated before, each task has an income and a current balance. The CPU scheduler has every task bid its current balance, i.e. all its money, for the CPU. Following the game theoretic model, the highest bidder wins the CPU at the price of the second highest bidder.

5.3 Bidding for I/O

The bidding for the disk is similar to the CPU, except the quanta size, the time to do the I/O, is of variable length. To account for this, each process bids a rate, the money per unit time, that they are willing to pay if they win the disk. The winning process is then charged at the rate of the second highest bidder. This allows the disk to maximize its money per unit time ratio.

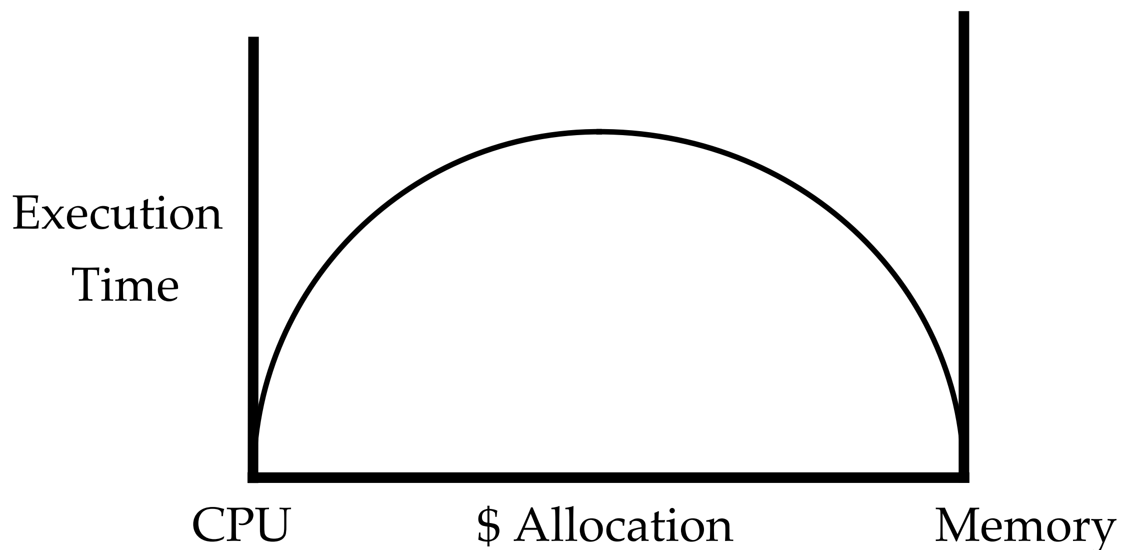
How a process determines its bid for the disk is slightly different than for the CPU. For the CPU, processes bid their full balance. For the disk, the processes bid using their balance

when the I/O would complete, because they receive their income at each time quanta while the disk is accessing their data. This gives a large advantage to higher priority tasks, because their income is greater.

5.4 Bidding for Pages in Memory

As you can well imagine, bidding for physical pages in memory is an auction for N pages for physical memory, and each task can win multiple pages. However, the difficult part is figuring out what to bid for those pages.

John Regehr pointed out to me that a task could spend all of its money on CPU, and would never complete a quanta because of its fault rate, or a task could spend all of its money on pages in memory and never complete a quanta because it would never get the CPU. So a process wants to optimize its spending on CPU and memory so that it competes the maximum number of quanta for its money.



That seems obvious, but what does a task bid for its Nth page in memory? It bids a prices for memory where it is optimal to have exactly N pages; if the price were less, the task would buy more pages, if the price was more, the task would accept fewer pages and spend more on the CPU. The problem is that these calculations involve knowing the future price of the CPU and I/O, and knowing the fault rate of a task with X pages in memory. In this model world, I was able to perform exact calculations of the future fault rates, but that would be impossible in a real system.

The prices for pages in memory is charged to each process at every time-step. The price charged is rebid every time a process faults, but does not change in between faults.

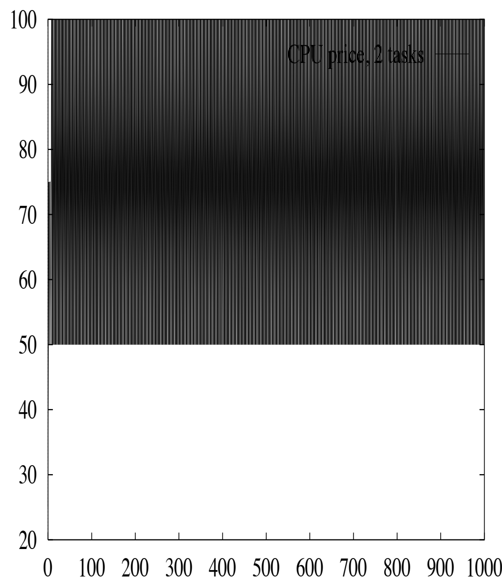
6.0 Results

A number of test were done to stress each particular scheduler, and then some general tests were done to check overall performance.

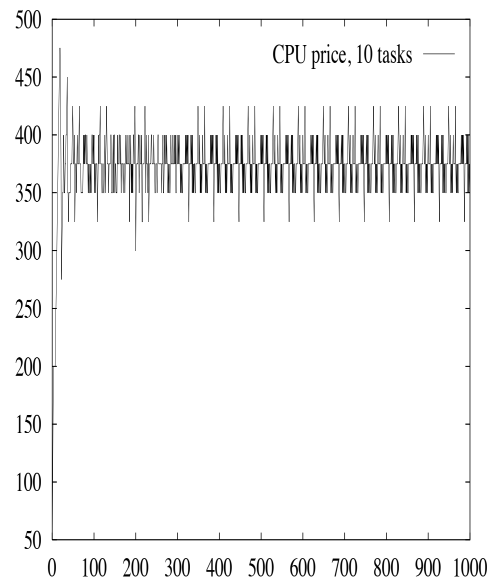
6.1 CPU Scheduler Results

Fair sharing of the CPU is achieved when the time spent on the CPU by each process divided by the priority of the process is equal for all processes. Start-time fair queuing makes a guarantee of fairness: the difference between the execution time of each process and the ideal fair execution time is bounded. The economic-based method for CPU allocation can make no such guarantee.

We can say that if the price remains the same for any period of time, we can make the same guarantee as SFQ for that period. In practice, the price of the CPU can fluctuate wildly, but the more processes that are competing for the CPU, the more stable the price becomes.



CPU Price over time with 2 tasks bidding



CPU Price over time with 10 tasks bidding

In the above-left graph, two tasks of priority 25 and 50 are bidding. The bidding cycles between 100 and 50, producing an average bid of 75 with a standard distribution of 25, one-third of the average bid! In the above-right, there are 10 tasks bidding, 5 of priority 25 and 5 of priority 50. The average bid is 375 with a sigma of about 12.5, roughly 3% of the average bid. So, the more bidders involved, the more stable the price is, and the more fair the sharing of the CPU.

Another piece of data that demonstrates this effect is the table of effective priorities. In the table, the nominal priority is the priority of the task, the effective priority is the portion of

the CPU that the task received, normalized to the other tasks running at a 50 priority. The second column has the effective priority of the task run against 1 other task with priority 50. The second column has the effective priority of the task run against 2 other tasks with priority 50. Each simulation was run for 10,000 time steps.

TABLE 1. Effective Priorities

Nominal Priority	Effective Priority bidding with 1 task	Effective Priority bidding with 2 tasks
50	50.000	49.993
45	35.063	45.709
40	39.993	39.997
35	32.142	34.370
30	37.474	30.174
25	49.960	24.984
20	24.974	19.560
15	14.276	14.718
10	9.088	9.360
5	4.042	4.341

This table demonstrates that with only two tasks competing for the CPU, the CPU's price variance is high enough to cause a big change in the effective priority of a task. The third column shows that all tasks' effective priority was within 1 of their nominal priority when competing with two other tasks. On the other hand, Start-time Fair Queuing guarantees that the effective priority of a task would be within 0.01 of the nominal priority after 10,000 time steps.

In conclusion, the CPU scheduler does achieve fair sharing of the CPU when there are lots of processes bidding for the CPU. When only two processes are bidding for the CPU, the CPU price can fluctuate wildly, allowing unfair sharing of the CPU. Three processes seems to be a minimum to keep the relatively stable, but the more processes there are, the more stable the price becomes, and the more fair the sharing becomes. However, SFQ was much fairer than this method, and is likely to be more fair no matter how many processes are present.

6.2 The I/O Scheduler

To test the I/O scheduler, 8 tasks were created which never faulted and always performed I/O if they received the CPU. We looked at two cases: one where all the priorities were the same, and one where one task had a priority 10 times the priority of all the other tasks.

Since the SCAN algorithm does not look at priorities in its scheduling, the conventional schedulers results will be the similar for both cases.

TABLE 2. Number of I/O completed by each task, all equal priority

Task #	Economic Scheduler	SCAN Algorithm
1	2003	2517
2	2070	2421
3	2013	2430
4	2018	2462
5	2030	2428
6	1996	2372
7	2061	2356
8	2016	2319
AVERAGE	2025.9	2413.1
SIGMA	24.85	58.70

The SCAN algorithm's throughput was 19% higher than the economic scheduler. It also had a much higher variance than the economic scheduler.

TABLE 3. Number of I/O completed by each task, task 1 having 10x priority

Task #	Economic Scheduler	SCAN Algorithm
1	5531	2457
2	1112	2442
3	1107	2464
4	1126	2446
5	1145	2400
6	1127	2376
7	1137	2370
8	1104	2305
AVERAGE	1673.6	2407.5

As mentioned above, SCAN does not use priorities, so its values are relatively unchanged. The overall throughput of the economic scheduler went down, but the throughput of the high priority task is 5 times the other tasks, and 2 times the throughput of the high priority task for the SCAN algorithm.

So, the economic scheduler had a much lower throughput than the conventional scheduler. It did, however, exceed it for the throughput of a single high priority process. Also, the economic scheduler had half the variance of the conventional scheduler.

6.3 Memory Allocator

To test the memory allocator, 8 processes were run in a tight amount of memory. Each process had a working set of 5 pages, a sloping factor of .50, and never performs I/O. 64

physical pages are available, which divided evenly among the tasks, would give each task a fault rate of 25%.

TABLE 4. Quanta completed and number of faults

Task #	Quanta Completed Economic	Quanta Completed LRU	# of faults Economic	# of faults LRU
1	6515	8719	968	1131
2	6718	8831	937	1120
3	6984	9270	962	1120
4	6699	8988	966	1125
5	6668	9088	937	1114
6	6919	8886	962	1099
7	6669	8413	960	1143
8	6525	7932	970	1127
AVERAGE	6712.1	8765.9	957.8	1122.4

Yes, this table does look odd; the LRU system has a much higher fault rate, but its processes completed more quanta. How can that be? The reason is that processes in the LRU system sometimes have more than the average number of pages, and sometimes have less. When they have more than the average, the fault rate drops dramatically, and the process runs quickly for a while. When the process has fewer number of pages than the average, its fault rate goes up dramatically, and the process faults a lot.   The economic scheduler is much more stable in its distribution of pages, while ends in predictably mediocre performance. The overall result is that the null process is running 38.6% of the time for the economic scheduler is running only 20.9% for the LRU scheduler.

TABLE 5. Quanta completed and number of faults, Task 1 has 10x priority

Task #	Quanta Completed Economic	Quanta Completed LRU	# of faults Economic	# of faults LRU
1	70409	14753	643	1169
2	2398	7669	774	1093
3	2511	8075	783	1093
4	2589	8093	769	1113
5	2583	7632	735	1095
6	3045	7267	598	1110
7	3370	7813	518	1082
8	3401	7101	464	1094
AVERAGE	11288.3	8550.4	660.5	1106.1

In this run, the priority for task 1 has been bumped up to 10 times the priority of each of the other tasks. Bumping up the priority has a huge effect in the economic system, but only a very small one in the conventional system. Task 1 has 10 times the priority of all the

other tasks in the economic system, but runs 25 times faster. For the conventional system, task 1 does not even run twice as fast as the average low priority process. Another change is the null process, now only running 4.4% of the time in the economic system, but running 22.7% under the LRU scheduler.

The dramatic change in the economic system is a result of many things. First, the high priority task holds more pages in memory, and, therefore, has an extremely low fault rate, while all the low priority process have a higher fault rate because they now have fewer pages. A second factor is that when the high priority task does fault, the I/O scheduler handles the fault quickly, allowing the high priority task to continue quickly. The last effect is that because the high priority task is almost always ready to run, and all the low priority tasks are faulting more often, and getting slower service on those faults, the high priority task is often the only task able to run, so it is not forced to share the CPU.

This multiplying effect is not seen in the conventional system, because only the CPU scheduler is affected by priorities. The LRU memory system still contains some variability to it, so even the high priority process sometimes has fewer pages than the average. Lastly, the high priority task is not preferred by SCAN, so it waits just as long a low priority task. So, in the end, the high priority task runs only slightly better than a low priority one.

6.4 Everything All Together

So far, we've looked at each scheduler individually, but nothing every operates in a vacuum. For this test, we looked at 8 processes, each with a working set of 5 and a sloping factor of 50%. In this case, however, there are 80 pages of physical memory to be used for paging. Lastly, there is now a 4% chance of a task doing I/O.

TABLE 6. Normal Test

Task #	Economic Scheduler	Conventional Sched.
1	11507	10921
2	11460	11013
3	11461	11513
4	11190	11046
5	11346	11345
6	11290	11116
7	11194	11231
8	11416	11715
AVERAGE	11358	11237.5
Sigma	115.4	254.3

As you can see, the economic scheduler had a higher average execution time and a lower variance among execution times. Since the execution time of the Null process was roughly the same on both schedulers, I can safely say that the higher execution time among the user processes was due to fewer faults in the economic scheduler.

The lower variance is harder to explain. I believe that the variance is lower due to balancing factors within the economic system. When a process receives ‘poor’ service, like receiving few pages in memory, it usually is compensated for by having a more money to spend later. The conventional system is just three random systems hooked together, and there is no way for any one scheduler to make up for the poor service of another one.

TABLE 7. Normal Test, task 1 has 10x priority

Task #	Economic Scheduler	Conventional Sched.
1	52994	27778
2	5473	8742
3	5481	9133
4	5338	8704
5	5438	9058
6	5774	8734
7	5829	8919
8	5774	8904

Once again, we look at the same test with one process with 10 times the priority of the other processes. The economic scheduler had the high priority process run 9.48 time the speed of the average of the low priority tasks, while the conventional scheduler only managed to run it 3.126 times faster. The obvious reason for the lack of performance by the conventional system is that only the CPU scheduler knows about priority. The memory system gives the high priority process a fault rate similar to all the other processes. In the economic system, the memory system gives the high priority process an extremely low fault rate.

TABLE 8.

Task #	I/O rate	WS size	slope f.	Economic Sched.	Conventional S.
1	4	5	.5	34443	32970
2	4	5	.75	29106	33005
3	20	5	.5	15655	13678
Null	N/A	N/A	N/A	13474	13075

This last example is here to show some of the balancing features of the economic system. The three processes are all different: the first is the standard process used in the two previous test, the second has a higher sloping factor, and the last has a higher I/O rate. All the processes have the same priority. In the conventional scheduler, the first two processes run about equal, but in the economic scheduler, the second process spends more on memory and receives less of the CPU. In both, the third process lags behind slowed down by the I/O. In the economic system, each process pays for the extra resources it uses, memory or I/O, and pays for it in execution speed. The conventional system does not penalize processes for grabbing more resources than another.

7.0 Conclusion

The economic based schedulers seem to be horrible. The CPU scheduler is reasonable with 3 process, but needs dozens of processes to be truly fair. The I/O scheduler is a modified shortest-seek-time-first algorithm and performs poorly under a heavy loading. Lastly, the memory system distributes pages evenly, but LRU out performed it precisely because LRU did not distribute pages evenly. In each test of an individual scheduler, the economic based schedulers performed poorer than the conventional scheduler.

However, tests of the overall systems seemed to favor the economic scheduler. It completed more quanta per a process than the conventional scheduler because it suffered fewer faults. It was more fair in terms of quanta completed, even though its CPU scheduler is less fair than the conventional scheduler. When one process had a higher priority than the others, the economic scheduler gave it execution time nearly proportional to its priority while the conventional scheduler effective priority was 1/3 of its true priority. Lastly, the economic system seems to balance everything well, charging process for the additional resources used.

In conclusion, the economic schedulers did poorly by themselves, but fit well together. The conventional schedulers did well in their respective categories, but did not have the synergy of the economic system when put together. I think more work is indicated for the economic scheduler - adding a measure of look-ahead to the I/O scheduler, finding a way to make the memory allocator work without perfect prediction, and to make the CPU scheduler less computationally expensive. I also think an event-driven simulation would provide much better numbers than this simple time step simulation. I do believe the results to be indicative of the economic-based schedulers, which performed well together.

8.0 Bibliography

Ferguson, Donald, Yechiam Yemini, and Christos Nikolaou, "Microeconomic Algorithms for Load Balancing in Distributed Systems", Proceedings of the 8th International Conference on Distributed Computing Systems, Computer Society Press: Washington, 1988, p 491-99

Goyal, Pawan, Xingang Guo, and Harrick Vin, "A Hierarchiacal CPU Scheduler for Multimedia Operating Systems", Proceedings of the Second Symposium on Operating System Design and Implementation, 1996, p. 107-121

Silberschatz, A., J. Perterson, and P. Galvin, "Operating System Concepts: Third Edition", Addison-Wesley Publishing: Reading, MA, 1991

Sutherland, I. E. "A Futures Market in Computer Time", Communications of the ACM, June 1968, p449-51