Synchronization Variability

I will be talking in the frame of the homework assignment, in that I provide a detailed report on various synchronization tools in order to give Ginormous Data Inc. (GDI) a more informed choice on which algorithm to employ for its consumers.

In an attempt to design an algorithm to give faster results at the cost of some correctness, my boss asked me to look into different synchronization techniques to determine which one is good enough for our purposes. The methods I have tested include a total lack of synchronization and the use of the Java's atomic long array library.

Assuming a lack of knowledge on these two topics, I will take some time to explain just what kind of implementation I designed. The tests without synchronization is essentially the same algorithm we employ, with the key difference being the removal of the 'synchronized' keyword. This, in effect, removes the lock around the associated function, allowing multiple threads to access the function and mutate the same variables. The basic idea comes from instruction parallelism, where multiple instructions are processed at the same time. This is, of course, not safe, however I plan to see the measure of failure this method has, seeing how our customers care for a quick extrapolation of their data, albeit at the cost of accuracy. If this method demonstrates accurate 'enough' data, it may be a viable algorithm to use.

The more complicated implementation to discuss would be the method utilizing the atomic long library. This implementation does not change a local variable in its swap function, but rather a global counter shared across all threads. Following the structure of the original algorithm, the process of decrementing antecedes the increment operation. In addition, another global variable of type long[] is assigned a value during the decrement and increment operations for later use in the function current, which returns type long[]. From my understanding in the specification, I was to implement this type of synchronization with a Plain mode archetype in mind, since it was to my understanding that I was to use the atomic long library and no other. Plain mode is the weakest memory order mode, though it holds the least restrictions and constraints. In general, "stronger modes impose more ordering constraints" and see a reduction in potential parallelism in at least one category: commutativity, coherence, causality or consensus (Lea). Implementing opaque, release/acquire, or volatile would have required the use of the atomic reference library for the first two and the varhandle library for the last one. The other synchronization method available would be to use the keyword 'synchronized', which is the original algorithm. In the context of the tests without synchronization, Plain mode introduces multiple problems in that data accesses may fail or might come in different orders. However, we use the atomic long array library to introduce atomicity and correctness. The attempt was successful, as the tests using this library had the correct output every time, so the correctness was the same as before. From my understanding of Dominant Resource Fairness, this implementation is not DRF. To start with, all threads run the same task with the same required amount of resources to run those tasks. Second, there is no scheduler implemented to divvy up the total available resources. For all these reasons, I would conclude that this implementation is not DRF.

For the measurements, I decided to run 100 million swap functions across a varying amount of threads and array sizes. The resultant output was of the real time, user cpu time, system cpu time (these three in the point of view of the Linux kernel), real swap time (ns), and cpu swap time (ns). These tests were run on Linux servers 7 and 10. They have different cpu families as well as a different amount of cores and processors. For example, server 7 featured 32 processors with 8 cores each. Furthermore, the chips were of the family E5-

Sheet 1: *Linux Server 7*

Swaps	Threads	Array Size	Real Time(s)	User CPU(s)	System CPU	realswap(ns)	cpuswap(ns)	Error	
100000000	1	5	2.56	2.586	0.081	23.39	23.3703	Synced	
	1	69	2.587	2.628	0.076	23.5734	23.5538		
	1	100	2.59	2.62	0.081	23.6521	23.6337		
	8	5	49.141	2m39.969	17.289	3912.25	1768.7		
	8	69	36.173	1m43.511	13.27	2875.24	1163.79		
	8	100	38.355	1m47.895	14.025	3049.71	1215.04		
	21	5	38.78	1m57.811	13.764	8091.31	1311.95		
	21	69	33.651	1m34.210	11.934	7016.51	1057.38		
	21	100	33.72	1m31.844	13.016	7029.96	1044.31		
	40	5	38.78	1m57.811	13.764	8091.31	1311.95		
	40	69	33.651	1m34.210	11.934	7016.51	1057.38		
	40	100	33.72	1m31.844	13.016	7029.96	1044.31		
	1	5	2.131	2.147	0.084	18.9334	18.9124	Unsynced	
	1	69	2.04	2.058	0.08	18.1463	18.1274		
	1	100	2.053	2.1	0.067	18.3327	18.3133		
	8	5	4.608	33.989	0.099	350.378	337.527		-21336
	8	69	5.019	37.991	0.082	382.83	337.336		-23264
	8	100	4.657	35.332	0.083	355.079	350.713		-15417
	21	5	4.62	1m30.035	0.098	919.956	897.729		-10209
	21	69	4.014	1m18.406	0.113	795.953	781.516		-16939
	21	100	3.701	1m12.053	0.103	728.691	717.724		-46982
	40	5	3.888	1m49.973	0.124	1465.93	1096.99		-26761
	40	69	3.697	1m43.878	0.0135	1381.98	1035.95		-21220
	40	100	3.165	1m27.379	0.0184	1198.68	872.069		-34692
	1	5	4.263	4.19	0.091	38.601	38.2068	AcmeSafe	
	1	69	3.736	3.772	0.072	35.4046	35.3796		
	1	100	3.808	3.894	0.072	36.6013	36.5772		
	8	5	33.047	1m26.254	12.53	2632.71	984.746		
	8	69	41.456	1m44.41	16.451	3306.01	1205.9		
	8	100	45.834	1m58.441	17.838	3645.83	1358.01		
	21	5	32.234	1m23.401	13.982	6735.35	970.232		
	21	69	36.652	1m32.961	14.292	7647.29	1068.31		
	21	100	34.401	1m30.482	13.247	7192.92	1034.56		
	40	5	36.457	1m35.308	13.057	14485	1079.21		
	40	69	43.59	1m50.110	16.21	17338.9	1258.35		
	40	100	48.868	2m7.334	17.688	19451.7	1445.27		
E5-2640	8 cores	32 processors	lnxsrv07	2 GHz					

2640 clocking in at 2 GHz. Linux server 10 on the other hand, was Intel Xeon as well but of the model Silver 4116 at 2.1 GHz. In addition, there were a total of 4 processors with 4 cores each. Other variables to consider are the different server loads, as there are other programmers running the same servers and testing their very own implementations. Also, I achieved my results after testing each set of parameters only once, so of course, your mileage will vary.

Moving forward and in light of the recent testing, I would like to conclude that utilizing the atomic long library holds little value over using our original synchronized method. As you can see in the results for both Linux servers, the AcmeSafe results for real time, user cpu time, and system cpu time are generally comparable or worse than the synchronized results. Though the values are quite similar to each other, AcmeSafe consistently outputs slightly worse times. In addition, the swap function took over twice as long in the worst case and 1.5 times in the best case. These values come from the measured real time of Linux server 7. In Linux server 10, the difference is much less exaggerated, though still noticeable. In that server, the best case and worst case for AcmeSafe in terms of synchronized was 1.5 times and twice as long, respectively. One thing that really surprised me, however, was the difference between testing on Linux servers 7 and 10. If you'll notice the time difference, the safe algorithms always reach around a minute and thirty of user cpu time (according to the Linux kernel) when handling 8 threads or more. Even the synchronized method reached an abnormally long time in Linux server 7, reaching times comparable to the safer synchronized methods. However, in real time this was no more than a few seconds.

Although real time is what matters to us in our measurements, the measured synchronous times across the two servers also begs investigation. For starters, Linux server 7 had times within 30 to 50 seconds for both groups; however, Linux server 10 demonstrated about 5 seconds for the original method and about 6 for the one I implemented. Naturally, I could chalk it up to server stress and the amount of people testing on the server, however it seems strange to me as Linux server 7 appears to have much better specs than server 10. In addition, server 10 is the required server where programmers must test their code while server 7 was simply one of the four choices available to us. For those reasons it seems odd to me to have such a massive discrepancy but testing at other points of the day yielded similar results. Those other points may have been unlucky as well, however, there is no sure way to tell.

Sheet 2: *Linux Server 10*

Swaps	Threads	Array Size	Real Time(s)	User CPU(s)	System CPU	realswap(ns)	cpuswap(ns)	
10000000		1	5	1.778	1.764	0.056	16.4933	16.4831 Synced
		1	69	1.835	1.839	0.044	17.0659	17.0515
		1	100	1.828	1.836	0.046	17.0648	17.047
		8	5	4.932	5.766	0.155	384.428	57.4485
		8	69	4.76	5.81	0.29	371.133	59.0734
		8	100	4.869	5.665	0.196	380.005	56.796
		21	5	5.07	5.0989	0.199	1038.23	59.9881
		21	69	4.718	5.514	0.194	963.618	55.0804
		21	100	4.958	5.993	0.214	1015.52	60.0915
		40	5	5.013	6.11	0.186	1957.55	60.9847
		40	69	4.922	5.702	0.154	1918.06	56.5219
		40	100	4.801	5.416	0.152	1868.36	53.5164
		1	5	1.343	1.321	0.054	12.092	12.0757 Unsynced
		1	69	1.355	1.357	0.04	12.2289	12.2028
		1	100	1.331	1.327	0.055	12.1429	12.1176
		8	5	3.737	14.374	0.06	287.753	142.554 -6096
		8	69	3.889	15.021	0.069	300.827	149.093 -15867
		8	100	3.753	14.538	0.051	290.677	144.137 -17157
		21	5	3.894	15.048	0.063	788.983	149.176 -9448
		21	69	3.934	15.249	0.05	801.105	150.955 -4396
		21	100	3.718	14.371	0.053	755.803	142.368 -5190
		40	5	3.803	14.621	0.094	1467.65	145.103 982
		40	69	4.141	15.984	0.069	1599.96	158.473 -23744
		40	100	3.652	14.169	0.057	1413.24	140.23 -21841
		1	5	3.235	3.213	0.063	30.9333	30.9132 AcmeSafe
		1	69	3.165	3.176	0.059	30.4586	30.435
		1	100	3.117	3.127	0.06	29.9685	29.9501
		8	5	6.702	7.107	0.217	525.768	71.3838
		8	69	6.711	7.171	0.183	527.343	71.475
		8	100	6.768	7.19	0.196	530.951	71.8726
		21	5	6.683	7.065	0.118	1375.72	69.6705
		21	69	6.792	7.333	0.243	1397.94	73.713
		21	100	6.79	7.293	0.201	1401.01	72.8265
		40	5	6.744	7.277	0.176	2644.67	72.2605
		40	69	6.817	7.347	0.147	2672.09	73.6439
		40	100	6.814	7.373	0.16	2673.58	73.1669
Silver 4116	4 cores	4 processors	lnxsrv10	2.1 GHz				

As for the discussion for the unsynchronized methods, I must preface the following details that the biggest percentage my tests were off by in both servers were 0.04%. This is obtained by using the magnitude of the most incorrect result (-46892) and dividing the result by the amount of swap operations performed, which was 100 million. There is a decrement and increment to the swap function, however as a whole the sum should equate to zero. In my eyes, then, the swap function failed at least 46892 times in the worst case. The lowest inaccuracy, 982, was off by 0.001%. For my tests specifically, the range of inaccuracies ranged from 0.001-0.04%. If this range of inaccuracy can satisfy our customers, then the unsynchronized method may be the most advisable algorithm to extrapolate data. Seeing how the unsynchronized method runs for at most a few seconds where the others take either a few more than that or a whole half minute more, this algorithm seems most unaffected by server factors.

References

Boehm, Hans J. "You Don't Know Jack about Shared Variables or Memory Models." *You Don't Know Jack about Shared Variables or Memory Models - ACM Queue*, ACM, 28 Dec. 2011, queue.acm.org/detail.cfm?id=2088916.

Ghodsi, Ali, et al. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types." University of California, Berkeley, 2011.

Lea, Doug. *Using JDK 9 Memory Order Modes*, 16 Nov. 2008, gee.cs.oswego.edu/dl/html/j9mm.html.