THRASHING

Peter J. Denning, Naval Postgraduate School, Monterey, California

January 2008
Rev 6/1/08

Abstract: Thrashing is an unstable collapse of throughput of a system as the load is increased.

Keywords: operating systems, virtual memory, multiprogramming, working set, contention resolution, Ethernet, packet switched multiplexing

Thrashing is an unstable collapse of throughput of a system as the load is increased. It violates the intuition that throughput should increase smoothly toward a saturation level as load increases. Thrashing occurs when contention for a critical resource prevents most jobs from actually using the resource.

Thrashing was first observed in the first-generation multiprogrammed virtual memory computing systems of the 1960s. Designers expected system throughput to increase toward a saturation level as the load (level of multiprogramming) increased. To their great surprise, throughput dropped suddenly after a critical load. (See Fig. 1.) Moreover, throughput would not return to its former high until the load was reduced below the critical value that triggered the thrashing -- a form of hysteresis. The system did not crash, but it did slow to an imperceptible crawl. The engineers called it “paging to death” because all the jobs were constantly queued up waiting for the paging disk to satisfy their page faults.

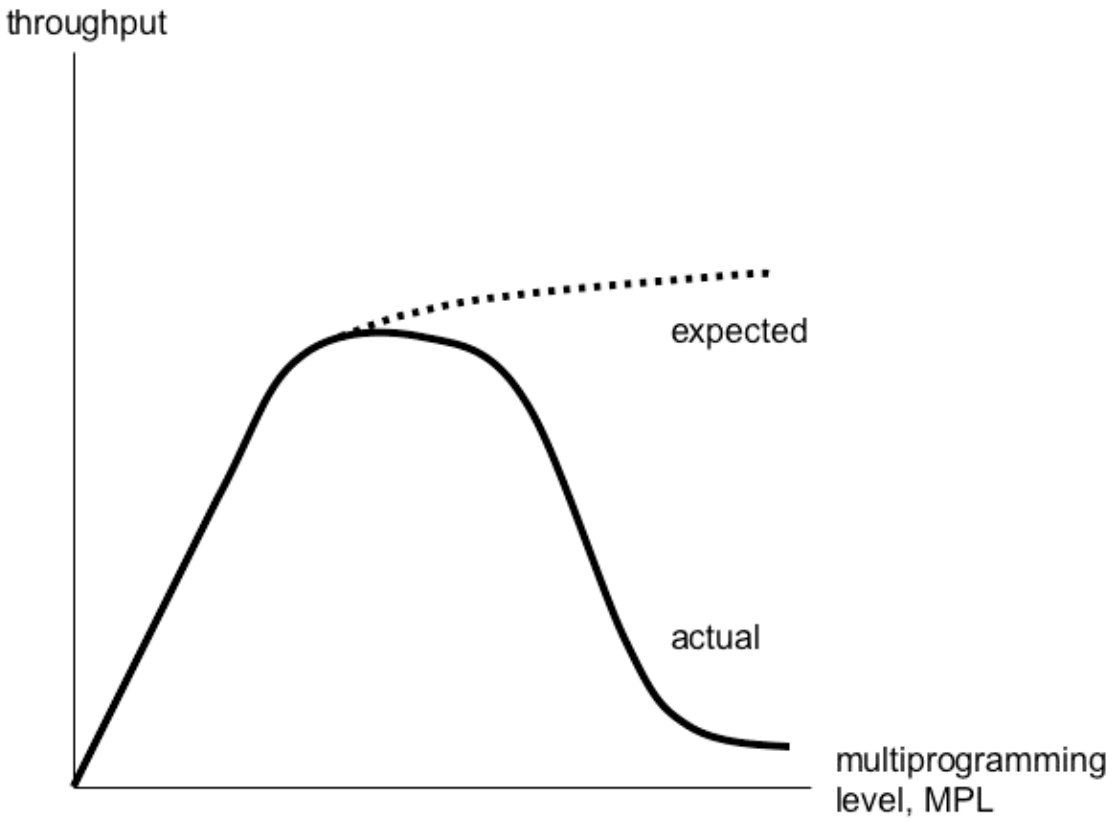

Figure 1. Thrashing in a multiprogrammed computer system.

Thrashing was a serious problem. It made the new generation of operating systems look like multi-million-dollar liabilities. Who would purchase such a system?

Thrashing was explained in 1968 (1). Increasing load in a fixed size memory causes the average partition to decrease, forcing an increase in the rate of page faults. The load at which the mean CPU time between page faults equals the disk service time is a tipping point: At higher loads, most jobs are queued at the paging disk, which becomes the system bottleneck.

As an example, consider a job that requires 1 second of CPU time on a system with a page fault service time of 10 milliseconds. At a small load, the job gets a large partition and generates (say) 20 page faults, which require a total of 0.2 seconds of disk time; this job's CPU efficiency is $1/(1 + 0.2) = 0.83$. At an intermediate load, the job's partition has been squeezed a little and it generates (say) 100 page faults; the CPU efficiency drops to 0.5. At a large system load, the job's partition has been squeezed a lot and it generates (say) 1000 page faults; its efficiency drops to 0.09. At the small loads, the job is CPU-bound; at the large loads, it is disk-bound. At the tipping point (efficiency 0.5), the average CPU time between page faults equals the disk service time.

The solution to thrashing is a load controller that allows all jobs enough memory space to keep their CPU efficiencies at 0.5 or higher. To accomplish this, a load controller separates submitted jobs into two sets: active and waiting. The active jobs are allowed to hold space in main memory. All other jobs are waiting. The working-set criterion is the ideal for deciding when to activate a waiting job. A job's working set is the set of pages required to allow the job to have a CPU efficiency above 0.5. If all active jobs have efficiencies above 0.5, the system will be below its thrashing tipping point. As long as every job gets enough space for its working set, it is impossible to activate enough jobs to push the system past its thrashing tipping point (2). The working-set criterion was found empirically not only to be optimal but to be more robust than other load-control criteria (3).

Within a decade after these early analyses, queueing network models were routinely used to quantify the relationship between throughput and the total demands for devices, and to help design load controllers to prevent thrashing (4).

Thrashing has been observed in other systems as well. Early packet networks provided examples. In unregulated networks, many servers vie for bandwidth in a common medium such as a satellite channel or Ethernet cable. When a server has a packet to transmit, it transmits into the medium. If another server jams it before completing, it stops transmitting, and tries again. This simple protocol works well for low loads, and system throughput increases with load. However, when the load gets high enough, there is a good chance that two or more servers will get into a loop where they start to transmit, detect a jam, stop, and repeat. Thereafter no one gets to transmit and throughput suddenly drops. This behavior was first observed in the ALOHA satellite communication network in the late 1960s (5).

Again, a properly designed load controller prevents the thrashing. In this case, the control is on the time interval from when a server drops out (it detected a jam) until it retries. It picks a random number for the first wait period; if that gets jammed, it waits twice as long; if that gets jammed, it waits twice as long again; and so on. This increasing-backoff protocol is used in Ethernets and cell phone networks.

Another instance of thrashing occurs from lock contention in database systems. A transaction requests and locks all records it needs, but if a requested record is already locked by another transaction, it releases all its locks and tries again. Again, the doubling of successive backoff intervals prevents thrashing, a condition in which lock contention prevents anyone from locking anything.

The common features of these systems are as follows: (*1*) A shared resource for which there can be many requests, and (2) a contention resolution protocol whose delay increases with the number of contenders. As the number of contenders increases, more and more time is spent on contention resolution and few jobs get through to the resource.

BIBLIOGRAPHY

1. P. J. Denning, Thrashing: Its causes and prevention, *Proc. AFIPS Fall Joint Comput. Conf.* 1968; **32**: 915-922.

2. P. J. Denning, The working set model for program behavior, ACM *Commun.* 1968; **11** (May): 323-333.

3. P. J. Denning, Working sets past and present, *IEEE Trans. Softw. Eng.*1980; **SE-6** (January): 64-84.

4. P. J. Courtois, *Decomposability*. Reading, MA: Academic Press, 1977.

5. L. Kleinrock, *Queueing Theory*, vol. 2. New York: Wiley, 1976, pp. 360-407.

FURTHER READING

A. Tanenbaum, *Modern Operating Systems*. Englewood Cliffs, NJ: Prenctice-Hall, 2003.