DOI:10.1145/3126494

Peter J. Denning

The Profession of IT
Multitasking Without Thrashing

Lessons from operating systems teach how to do multitasking without thrashing.

Our individual ability to be productive has been hard stressed by the sheer load of task requests we receive via the Internet. In 2001, David Allen published *Getting Things Done*,[1] a best-selling book about a system for managing all our tasks to eliminate stress and increase productivity. Allen claims that a considerable amount of stress comes our way when we have too many incomplete tasks. He views tasks as loops connecting someone making a request and you as the performer who must deliver the requested results. Getting systematic about completing loops dramatically reduces stress.

Allen says that operating systems are designed to get tasks done efficiently on computers. Why not export key ideas about task management into a personal operating system? He calls his operating system GTD, for Getting Things Done. The GTD system supports you in tracking open loops and moving them toward completion. It routes incoming requests to one of these destinations in your filing system:

- Trash
- Tasks that might one day turn out to be worth doing
- Tasks that serve as potential future reference points
- Tasks delegated to someone else, awaiting their response
- Tasks that can be completed immediately in under two minutes
- Tasks accepted for processing

The first four destinations basically remove incoming tasks from your workspace, the fifth closes quick loops, and the sixth holds your incomplete loops. GTD helps you keep track of these unfinished loops.

The idea of tasks being closed loops of a conversation between a requester and a perform was first proposed in 1979 by Fernando Flores.[5] The "conditions of satisfaction" that are produced by the performer define loop completion and allow tracking the movement of the conversation toward completion. Incomplete loops have many negative consequences including accumulations of dissatisfaction, stress, and distrust.

Many people have found the GTD operating system to be very helpful at completing their loops, maintaining satisfaction with work, and reducing stress. It is a fine example of us taking lessons from technology to improve our lives.

Multitasking

Unfortunately, GTD does not eliminate another source of stress that was much less of a problem in 2001 than today. This is the problem of thrashing when you have too many tasks in progress at the same time.[2]

The term multitasking is used in operating systems to mean executing multiple computational processes simultaneously. The very first operating system do this was the Atlas supervisor, running at the University of Manchester, U.K., in 1959. IBM brought the idea to the commercial world with its OS 360 in 1965.

Operating systems implement multitasking by cycling a CPU through a list of all incomplete tasks, giving each one a time slice on the CPU. If the task does not complete by the end of its time slice, the OS interrupts it and puts it on the end of the list. To switch the CPU context, the OS saves all the CPU registers of the current task and loads the registers of the new task. The designers set the time slice length long enough to keep the total context switch time insignificant. However, if the time slice is too short, the system can significantly slow down due to rapidly accumulating context-switching time.

When main memory was small, multitasking was implemented by loading only one task at a time. Thus, each context switch forced a memory swap: the pages of the running task were saved to disk, and then the pages of the new task loaded. Page swapping is extremely expensive. The 1965 era OSs eliminated this problem by combining multitasking with multiprogramming: the pages of all active tasks stay loaded in main memory and context switching involves no swapping. However, if too many tasks were activated, their allocations would be too small and they would page excessively, causing system throughput to collapse. Engineers called this thrashing, a shorthand for "paging to death."

Eventually researchers discovered the root cause of thrashing and built control systems to eliminate it—I will return to this shortly.

Figure 1. In this memory map of a Firefox Browser in Linux, the colored pixels indicate that a page (vertical axis) is used during a fixed size execution interval (horizontal axis). The locality sets (pages used) are small compared to the whole address space and their use persists over extended intervals.

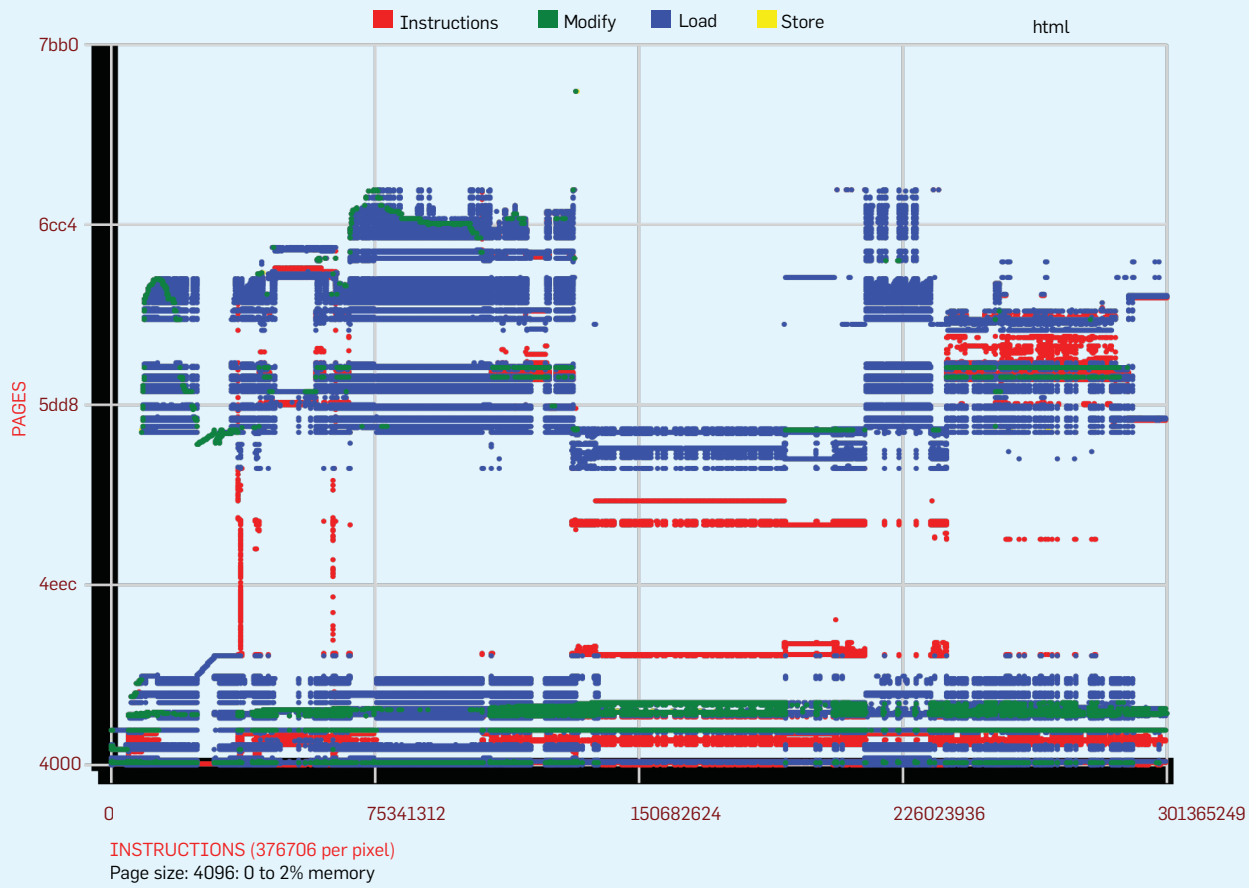


FIGURE 1 COURTESY OF ANDRIAN MCMENAMIN

Human Multitasking

Humans multitask too by juggling several incomplete tasks at once. Cognitive scientists and psychologists have studied human multitasking for almost two decades. Their main finding is that humans do not switch tasks well. Psychologist Nancy Napier illustrates with a simple do-it-yourself test.[7] Write "I am a great multitasker" on line 1 and the series of numbers 1, 2, 3, ..., 20 on line 2. Time how long it takes to do this. Now do it again, alternating one letter from line 1 and one numeral from line 2. Time how long it takes. For most people, the fine-grained multitasking in the second run takes over twice as long as the one-task-at-a-time first run. Moreover, you are likely to make more errors while multitasking. This test reveals just how slow our brains are at context switching. You can try the test a third time using time-slicing, for example writing five letters and then switching to write five numerals. With fewer context switches, time-slicing is faster than fine-grained multitasking but still slower than one-at-a-time processing.

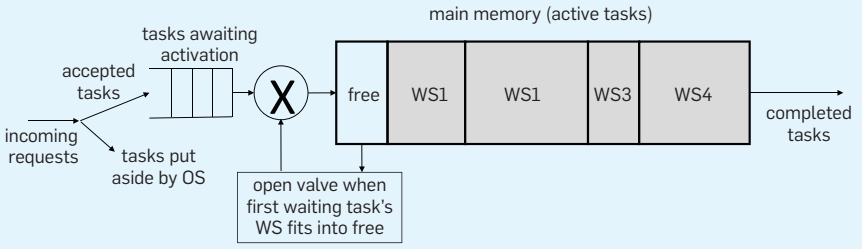
Human context switching is more complicated than computer context switching. Whereas the computer context switch replaces a fixed number of bytes in a few CPU registers, the human has to recall what was "on the mind" at the time of the switch and, if the human was interrupted with no opportunity to choose a "clean break," the human has to reconstruct lost short term memory.

Context switching is not the only problem. Whereas a computer picks the next task from the head of a queue, your brain has to consider all the tasks and select one, such as the most urgent or the most important. The time to choose a next task goes up faster than linear with the number of tasks. Moreover, if you have several urgent important tasks, your brain can get stuck in a decision process that can take quite a long time to decide—a situation known as the choice uncertainty problem.[4]

A third factor that slows human multitasking is gathering the resources necessary to continue with a task. Some resources are physical such as books, equipment, and tools. Some are digital such as files, images, sounds, Web pages, and remote databases. And some are mental, things you have to remember about where you were in the task and what approach you were taking to perform it. All these resources must be close at hand so that you can access them quickly.

These three problems plague multitaskers of all age groups. Many studies report considerable evidence of negative effects—multitasking seems to reduce productivity, increase errors, increase stress, and exhaust us. Some researchers report that multitaskers are less likely to develop expertise in a topic because they do not get enough inten-

Figure 2. OS control system to maximize throughput with variable partition of main memory determined by task working sets.



sive focused practice with it. Some fret that if we do not learn to manage our multitasking well, we may wind up becoming a world of dilettantes with few experts to keep our technology running.

Thrashing happens to human multitaskers when they have too many incomplete tasks. They fall into a mood of "overwhelm" in which they experience considerable stress, cannot choose a next task to work on, and cannot stay focused on the chosen task. It can be a difficult state to recover from.

Let us now take a look at what OSs do to avoid thrashing and see what lessons we can take to avoid it ourselves.

Locality, Working Sets, and Thrashing

The OS seeks to allocate memory among multiple tasks so as to maximize system throughput—the number of completed tasks per second.[3]

The accompanying Figure 1 is strong graphical evidence of the principle of locality—computations concentrate their memory accesses to relatively small locality sets over extended intervals. Locality should be no surprise—it reflects the way human designers approach tasks.

We use the term working set for OS's estimate of a task's locality set. The formal definition is that working set is the pages used in a backward-looking window of a fixed size T memory references. In Figure 1, T is the length of the sampling interval and the working set equals the locality set 97% of the time.

Each task needs a workspace—its own area of memory in which to load its pages. There are at least two ways to divide the total memory among the active tasks. In fixed partitioning, the OS gives each task a fixed workspace. In working-set partitioning, the OS gives each task a variable workspace that tracks its locality sets. Fixed partitioning is susceptible to thrashing as the number of tasks sharing memory increases because each gets a smaller workspace and, when the workspaces are smaller than the working sets, every task is quickly interrupted by a page fault.

Under working-set partitioning the OS sizes the workspaces to hold each task's measured working set. As shown in Figure 2, it loads tasks into memory until the unused free space is too small to hold the next task's working set; the remaining tasks are held aside in a queue until there is room for their working sets. When a task has a page fault, the new page is added to its workspace by taking a free page; when any page has not been used for T memory references, it is evicted from the task's workspace and placed in the free space. Thus, the OS divides the memory among the active tasks such that each task's workspace tracks its locality sets. Page faults do not steal pages from other working sets. This strategy automatically adjusts the load (number of active tasks) to keep throughput near its maximum and to avoid thrashing.

Context switching is *not* the cause of thrashing. The cause of thrashing is the failure to give every active task enough space for its working set, thereby causing excessive movement of pages between secondary and main memory.

Translation to Human Multitasking

Although the analogy with OSs is not perfect, there are some lessons:

- Recognize that each task needs a variable working set of resources (physical, digital, and mental), which must be easily accessible in your workspace. Analog: the working set of pages.
- Your capacity to deal with a task is the resources and time needed to get it done. Analog: the memory and CPU time needed for a task.
- Some tasks need to be held aside in an inactive status until you have the capacity to deal with them. Analog: the waiting tasks queue.
- When a task's working set is in your workspace, protect it from being unloaded as long as the task is active. Analog: protect working sets of active tasks and do not steal from other tasks.
- You will thrash if you activate too many tasks so that the total demand is beyond your capacity. Analog: insufficient CPU and memory for active tasks.
- If you are able to choose moments of context switch, select a moment of "clean break" that requires little mental reacquisition time when you return to the task. If you cannot defer an interruption to such a moment, you will need more reacquisition time because you will have to reconstruct short-term memory lost at the interruption. Analog: ill-timed interrupts can cause loss of part of a working set.

You are likely to find that you cannot accommodate more than a few active tasks at once without thrashing. However, with the precautions described here, thrashing is unlikely. If it does occur you will feel overwhelmed and your processing efficiency will be badly impaired. To exit the thrashing state, you need to reduce demand or increase your capacity. You can do this by reaching out to other people—making requests for help, renegotiating deadlines, acquiring more resources, and in some cases canceling less important tasks. C

References

1. Allen, D. *Getting Things Done*. Penguin. 2001.
2. Christian, B. and Griffiths. T. *Algorithms to Live By: The Computer Science of Human Decisions*. Henry Holt and Company, 2016.
3. Denning, P. Working sets past and present. *IEEE Trans Software Engineering SE-6*, 1 (Jan. 1980), 64–84.
4. Denning, P. and Martell, C. *Great Principles of Computing*. MIT Press, 2015.
5. Flores, F. *Conversations for Action and Collected Essays*. CreateSpace Independent Publishing Platform, 2012.
6. McMenamin, A. Applying working set heuristics to the Linux kernel. Masters Thesis, Birkbeck College, University of London, 2011; http://bit.ly/2vFSgY8
7. Napier, N. The myth of multitasking, 2014; http://bit.ly/1vuBGcC

Peter J. Denning (pjd@nps.edu) is Distinguished Professor of Computer Science and Director of the Cebrowski Institute for information innovation at the Naval Postgraduate School in Monterey, CA, is Editor of ACM *Ubiquity*, and is a past president of ACM. The author's views expressed here are not necessarily those of his employer or the U.S. federal government.