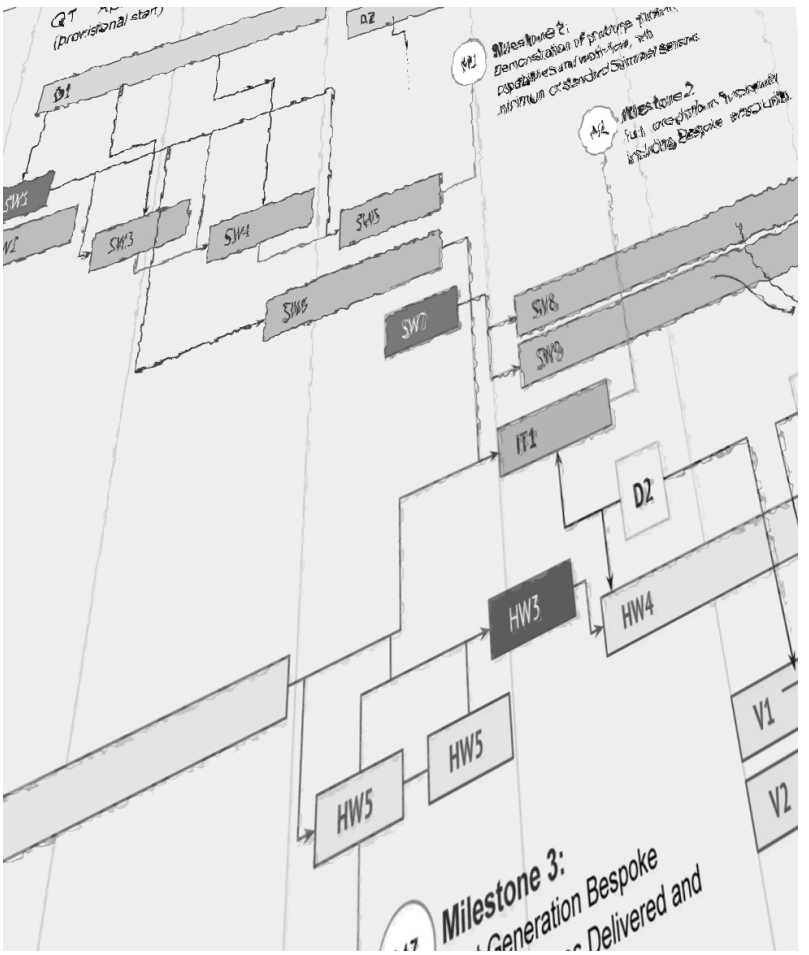
Workload Management

10



10.1 The task in hand

From previous sections, we have established that operating systems manage tasks, and that tasks can be classified as processes or threads. Some operating systems have their own terminology for these concepts which we will avoid for the moment. Instead, let us consider what these two types of tasks are allowed to do and what they are not.

Processes: are programs that run under control of the operating system, such that when a program begins, the OS allocates it a memory space that is private to that process. It also adds the program to a task-list, to allow it to be given regular intervals of CPU time to perform its activities.

One consequence of a process having its own private portion of memory space is that other processes cannot, in theory at least, access or alter that memory. This means that the concept of compartmentalisation of processes is achieved. This has many benefits, including privacy and security of data within a task, and resilience to being affected if another process crashes. Imagine, for example, if your banking app could be spied upon by another app simply by it looking at the other app's memory. A critical purpose of the operating system is to prevent this from happening.

Threads: Within a process there may be many threads. These are submodules, or mini-programs that each perform a purpose in their own right and which are required by the program designer to operate concurrently in order to perform a collective outcome associated with that process. Threads do not exist on their own, they must be created within a process.[131]

[131] Threads are sometimes referred to as lightweight processes, but they are not processes in the true sense.

A thread example might be encountered in a process which reads data from a microphone into memory, whilst also copying the resulting audio data from memory to disk. In order to get smooth and uninterrupted recording, it may be necessary for the process to run the microphone function as one thread, and the disk writing function as a second thread. Then both threads get a slice of CPU time at frequent intervals, and if correctly set up, it will allow recording and disk transfer to happen as if concurrently. Additional threads can be introduced, for example one that monitors and updates a GUI, without upsetting the operation of the first two threads.

Importantly, because threads are like siblings within the main process, they all use the same memory resources allocated to that process on

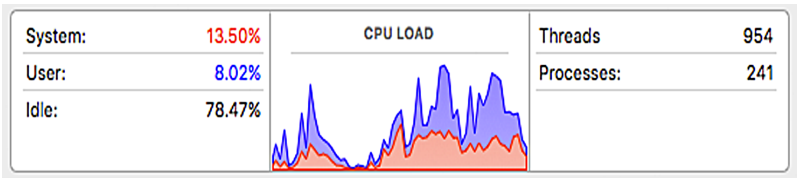


Figure 10.1: CPU workload process Monitor. This view shows a MAC OS performance monitor view, with both the kernel/system load (bottom trace) and the user load (upper trace).

equal terms. They can access blocks of memory within that process, and share information. If this happens inside a process then it is potentially advantageous, since it allows easy sharing of key data between threads. Generally, threads tend to work together on some level, therefore sharing data and memory accesses with the threads of their mutual encapsulating process makes sense.

In a typical desktop or laptop, a large number of processes may be running simultaneously, and each many have many associated threads. The work these processes and threads are doing can vary from one moment to another. This is observable in Figure 10.1 where we can see 241 processes are running, and 954 threads! This is all on a single laptop computer. Notice also that this monitoring tool displays system and user workloads as two distinct categories, and varying over a time window.[132] In this case the OS kernel is using more CPU time than the user on average, but overall, the CPU is idle nearly 80% of the time in this example, suggesting it is not being very heavily used by the user at this particular point.

[132] In this case the time window was of the order of a few minutes.

10.2 Process ID and housekeeping

In order to keep track of processes as they execute, the operating system makes use of two important concepts. The first is the idea of a **process identifier**, or **PID**. This is simply a unique number that is assigned to any process when it is started. This means that every process on the system has a unique PID.

Once a PID is assigned, the operating system can then keep a more detailed record of the process's key information, known as a **process control block** or **PCB**. A PCB contains a variety of key information about its associated process, including:

- Process State (starting, ready, waiting, etc).
- Information about its parent process,
- Task priority and privileges,

- IO attribute related information,
- General info such as CPU state data, held whilst the task is suspended.

Together, these pieces of information allow a special task management function, known as a **scheduler**, to manage each process according to the rules and configurations of the operating system, and the particular task attributes assigned to the process and its owner.[133]

[133] A process could belong to one of several users of the same system, or different login identities operated by the same user.

10.3 Thread management models

Although threads belong to processes, there are several models under which a thread can be created and managed. Figure 10.2 illustrates these two models, which can be described as follows:

User-level threads exist when multiple user processes exist in an operating system which are scheduled by the kernel, but in which the threads belonging to those processes are scheduled within the process code itself (the kernel does not see those threads as separate entities). This is achieved by a process having its own scheduler built into its software. The advantage here is that the threads within that process can be scheduled and managed in entirely custom fashion, within the boundaries of that process.

Kernel-level threads require processes to request that the kernel creates and manages threads on that process's behalf, and then allow them to be managed by the kernel scheduler. The advantage here is that processes do not need to include their own custom thread scheduling code as part of their software, but must also accept all of the constraints of the kernel scheduler, which may vary from one operating system to another.

In theory both schemes could coexist. The operating system has no visibility of user-level threads, it cannot see them, and therefore even in a kernel-level thread model, processes could also have their own local user threads too. However, in general, the idea is that where kernel level threading is available, programmers should utilise the operating system features accordingly. This might be an enforced requirement in some systems, to ensure all threads are managed according to a given policy.

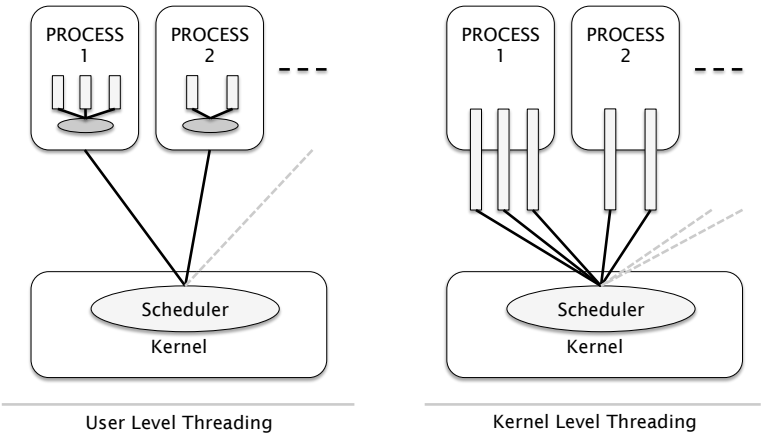


Figure 10.2: Thread management models. User threads are created and managed by additional schedulers that exist only within each process. Kernel level threads are created and managed by the kernel on behalf of processes.

10.4 Task scheduling

We have now gained an overview of how tasks can coexist in a machine's operating system, and how they appear to all intents and purposes to execute in parallel with each other. We have also learned that in actual fact this may be an illusion, and the vast majority of processes are executed by a method of interleaving small portions of the code execution of each task in turn.

A useful analogy is that of a reader who buys five books to take on holiday. They could take the conventional approach, and read one book at a time, completing each book before starting the next. This is analogous to serial workload management. In early computer systems, the computer could only process one job at a time, and could only run one process, and no threads. Users would submit their jobs to an operator who would schedule them manually for loading and running one by one.

As computer systems improved, and early operating systems progressed, it became possible to run multiple jobs using time-slicing, such that part of the CPU time was allocated to each job. This is essentially the same as task scheduling as we know it today, though the granularity has become much finer.

Returning to our analogy, the book reader could instead chose to read one chapter of each book in turn, such that all five books appear to be being read at the same time. This is yet another form of multiplexing, a concept we have already come across.

Now, imagine that the reader gets somewhat over-enthusiastic about this new method of reading, and decides to read a single page of each book in turn, and then just a single paragraph. The reader would spend more and more time physically swapping between books, replacing bookmarks, and so on, and less and less time actually reading. This is an example of the impact of the **task-switching overhead**. And what this tells us is that there is a limit to how fast tasks can be switched, and how fine the granularity of task switching can be, whilst maintaining reasonable efficiency.

In the worst case, any task switching will result in the jobs or tasks being executed in more absolute time than if they were executed serially without the switching overheads. However, there are reasons why this is not always true, and indeed it can be the case that task switching achieves much higher throughput (work done in a given amount of time) than the serial case. Of course it is also the case that we generally also want many tasks to run in concurrent fashion in a modern operating system, and therefore the option of serial execution is simply not feasible in a modern computing environment.

Regardless of how fine the granularity of the task switching, it must be managed by some sort of control function in the operating system kernel. This is known as the **task scheduler**, and this task scheduler follows some well-defined policy with respect to how it manages the tasks. These policies are known as **scheduling algorithms**.

10.5 Scheduling algorithms

The concept of task scheduling can be quite complex, with a number of fairly sophisticated algorithms. However, ultimately all of these approaches attempt to fulfil one basic purpose: to ensure that all processes receive an appropriate share of resources. If this can also be done in such a way that the system's resources are optimally utilised, then this is the ideal situation.

Let us first consider a very rudimentary task scheduler, an imaginary case that will allow us to grasp the concepts a little better. We will call it 'SimpleSched'.

This task scheduler has a list of currently running tasks, known as a **task list**. Each time a new program is started, it is added to the list, and each time a program ends, it is removed. The list can therefore grow and shrink dynamically as the system goes about its business.

SimpleSched will use a simple **round-robin** model of scheduling. This algorithm simply goes through the list one task at a time, and executes each task for a fixed period, say 1ms. Then, when it reaches the end of the list, it goes back to the beginning and starts over again, repeating indefinitely.

When the next task is picked from the list, the current running task is **suspended**, and the next task is **resumed**. This is like pressing pause and play on your TV recorder with different recordings. This switching mechanism is known as a **context switch**. Among other things, this involves copying the paused CPU state into the process control block (PCB) and restoring the resuming process's paused CPU state from its PCB back to its active condition in the processor.

We can see that SimpleSched has some notable properties. First of all, every task gets an equal amount of CPU time. Also, every task is executed once per full pass through of the list.

On one hand this algorithm represents fairness, which is an aspect of task scheduling that we may find desirable. However, there is a problem here that would become apparent in a real system fairly quickly. Because every task has to wait its turn in the list, important tasks are made to wait whilst unimportant ones have their turn at utilising the CPU. This means that SimpleSched has no sense of prioritisation. **Prioritisation** or **task priority** is the idea that some tasks are more important than others and might be given more CPU time, and/or executed more often.

There is a further problem with SimpleSched. Whilst it seems fair to give every task the same amount of CPU time, it is entirely possible that this is a huge waste of resources. Consider that one task may be very intensively processing data, and will use every clock-cycle of its allocated CPU time usefully, but another task may have very little to do, and it will effectively be idle for much of its allocated time.

Yet another task may want to do lots of work, but actually spends a lot of time waiting for something outside its control, such as data requested from a hard drive, which may well need 5 or 10 ms to become available. This kind of task is waiting for an IO event.

We might argue that actually, having equal time-slices is not fair after all: If some of those tasks do not need that much of the CPU's resources then why deprive other tasks from using that capability for no reason?

Fortunately, with the right task scheduling policies, we can attempt to overcome some of these limitations quite successfully. However, we first need to understand another scheduling concept: queues.

10.5.1 Task scheduling queues

In our first example, SimpleSched, we described a group of tasks organised as a list. Whilst this is a valid scenario for a scheduler, it doesn't conveniently encapsulate some of the ideas we wish to develop, and which more complex scheduling algorithms use. Instead, we need to make use of the concept of a **queue**.

Imagine that SimpleSched was modified so that instead of having a list that the scheduler works through one by one, it instead has a queue. As each task is suspended, it is placed at the back of the queue, and the item at the front of the queue is resumed for a period of time. Eventually every task in the queue will be resumed, and then suspended and pushed back into the queue to come around again. New tasks are fed into the end of the queue to join the group, and terminating tasks are simply dropped from the queue and not sent back into the queue for a further scheduling cycle.

In its simplest case, this is a **FIFO** queue (**First-In First-Out**). You might note that the net effect of this model appears to be identical to SimpleSched. It still sees each task executed once per full cycle of the queue. However, we will see shortly that adopting a queue model will give us new capabilities for scheduling. At this point, it is worth noting that we have actually described a basic model of scheduling, known as a **pre-emptive scheduler**.

10.6 Pre-emptive scheduling

So it turns out that if SimpleSched is modified to use a queue rather than a list, it is effectively a pre-emptive task scheduler as would be recognised by most operating systems.

A non-preemptive scheduler in contrast, would go through the queue or list, resuming each task in turn, and allowing it to run until that task yields control back to the scheduler. In effect, the tasks decide how long they get. But this means that tasks have to be very well-behaved in order to achieve a fair and sensible scheduling outcome.

The idea of pre-emptive scheduling is that a regular time-keeping element triggers the switch between the current and the next process. This is often achieved by a system component called a **Repetitive Interrupt Timer** (**RIT**) which sends a signal to the CPU after a certain amount of time has elapsed. In the simplest case, this time period is uniform for all

tasks. However, there is no reason why this must be the case, and some tasks may be allowed a longer period of execution than others.

If a task is assigned a rating, known as a **priority level**, the scheduler can apply different rules to that task, as compared to a task with another priority rating. For instance, it might decide if it should be given a short period of CPU time, a medium period, or a long period, for example.

Another feature that can be introduced into a scheduling policy is the idea of processes voluntarily yielding control back to the scheduler prematurely. Imagine a process that is resumed, only to decide that is has no work to do right now. It could use the time slice up by simply waiting. On the other hand, it could send an instruction back to the scheduler to say it has nothing to do: in other words, it **yields control** back to the scheduler, which can then immediately suspend the task and resume another one.

This concept allows the scheduler to use all of the otherwise unused time in idle tasks to service other tasks in the queue.[134]

[134] This is a point for the programmer to consider carefully when writing an application. Most programming languages provide support for this: for example pthread_yield() in the C language, and yield() in Java.

An important thing has just happened here, and it is worth noting. When a task scheduler can allocate fixed times per task, even if the amount of time is different for each task, then this allows the task scheduling to be **deterministic**. In other words, we can always predict how long a particular task has to wait before it next gets to be executed, and how much total CPU time it is able to utilise.

However, if we introduce the concept of tasks being able to yield control, then there is no easy way to know how long any task time-slice will last, and the determinism of the system becomes very weak, if not altogether eliminated.

In this case, whilst we still know the worst-case time between a task's successive time-slices, we do not know the best-case or even the average. This is a very important issue for some applications, such as real-time and safety-critical systems, where it will result in execution times of particular activities varying widely (a problem known as **jitter**).

We can consider an example with some numbers here. Suppose a simple scheduling algorithm requires 5ms per task switch, and allocates 20ms per time-slice. And consider that there are 100 tasks in the queue. How much time is available per thread ? How often do threads repeat?

Timeslices Example (simple case)

Each task requires a total of 25ms. There are 100 tasks so a total of 25 x 100 ms = 2500 ms is needed for all tasks to execute once.

A task receiving 20 ms of CPU time every 2500 ms has 20/2500 = 0.008 (0.8%) of CPU time.

We can also see that each task repeats once every 2.5 seconds, or a frequency of 1/2.5 = 0.4 times per second.

This is quite slow, and would not give very smooth system performance.

We can also consider what happens if **yield** is used to skip over inactive tasks in the queue. Suppose 50 of the tasks yield as soon as they are resumed, what is the result?

Timeslices Example (with yield)

Each active task requires a total of 25ms. Each inactive task yields as soon as the 5ms task-switch is completed. So we have (50 x 25) + (50 x 5) = 1500 ms total time

An active task receiving 20 ms of CPU time every 1500 ms has 20/1500 = 0.0133 (1.33%) of CPU time.

We can also see that each task repeats once every 1.5 seconds, or a frequency of 1/1.5 = 0.66 times per second.

Still relatively slow, but quicker, and CPU time per thread has increased.

10.6.1 Multiple queues

Given the scenario described, where a task may decide to yield control because it has no work to do, there is an opportunity to improve the scheduling model further by introducing multiple queues.

Consider what happens with the single queue: A task is resumed, it has nothing to do, so it yields control back to the scheduler and is put back in the queue for another round. So the scheduler could spend an

appreciable amount of time resuming processes that simply yield control back to the scheduler again straight away. This is costly: remember the task switching overhead.

A better approach, which helps to alleviate this problem, is having two or perhaps more queues. Then tasks can be placed in a queue that best represents their situation at that moment.

Let us consider a scenario where several queues are defined as follows:

- **Ready Queue:** A queue containing all tasks that are currently ready to be executed again.
- **IO Wait Queue:** A queue containing all tasks waiting for an IO activity to complete.

In this system, with two queues, the scheduler can move tasks to the **IO Wait Queue** when they yield and signal that they have an IO wait condition. This means that the task scheduler can suspend and resume tasks in the **Ready Queue**, knowing that they are not going to simply yield control again immediately after being resumed, or waste time waiting during their time-slice. Also, since all of those extra and unnecessary task-switch overheads can be eliminated, this leaves a little more CPU time for servicing ready tasks.

Meanwhile, the tasks that have ended up in the IO Wait Queue are made to wait in line, until they achieve their IO objectives, at which point such a task is moved back to the ready queue.

10.6.2 Interrupts and polling

We have just encountered the idea of a task being able to be designated as waiting for an IO event. In a sophisticated task scheduler, such a task will be moved to the IO-Wait Queue and will stay there until the IO event is completed. This raises the question of how the scheduler knows this is the case, and introduces the idea of polling versus interrupts.

IO-Polling is an algorithm in which a task repeatedly reads the status of a device to see if it has completed the requested task. For example, if a disk access is required, a thread could request a disk access, then repeatedly read the status of the device at short intervals. Of course the slowness of the disk access (of the order of 5-10ms) means that a lot of CPU time will be spent getting the answer 'no i am not ready yet'.

This approach is wasteful. As noted earlier, if we have a single-queue scheduler, every task must be resumed at a frequent interval, in order

for them to update their polling context, and some of those tasks will simply be spending their time-slices reading device statuses and getting the 'I am not ready yet' response.

Interrupts are signals used to indicate that events have occurred in a computer system. They can be driven by timers to create repetitive events, or to create delays, and also by IO devices and other hardware resources, to indicate occurrences such as a disk access being completed. Therefore, interrupts can be very useful in task scheduling.

In the more sophisticated task scheduler, we described moving tasks to the IO-Wait Queue when they were waiting for an IO event. Now if we add interrupts into the model we can see how this works. A thread wishing to perform an IO operation will set up the request such that when it is completed, an interrupt is generated. The thread can then put itself into a suspended (sleep) state until the interrupt is received.

In the case of a single queue model, the sleeping thread could simply yield its execution as soon as it is resumed, and this will continue to happen until it receives an interrupt.

In the case where an IO-Wait Queue is used, the interrupt causes the thread waiting in the IO-Wait queue to request that the scheduler moves the thread back to the ready queue, and this means that the waiting threads do not waste any CPU time at all whilst waiting for an event.

It is worth noting that events can be caused by disk units, by input devices such as cameras, by network interfaces, and many other IO devices and peripherals. For example, a thread might initiate a disk data read, and then have nothing else to do until the disk has read the data, transferred it to a designated area of memory using DMA, and generated an interrupt.

One of the consequences of having an interrupt-driven IO policy is that many tasks can share CPU time in such a manner that when tasks are stalled waiting for IO, other tasks can use the spare CPU capacity for something useful. These tasks may even belong to different users, meaning that other users get to use the spare system capacity. This approach should ensure that the utilisation of resources is as efficient as reasonably possible.

10.7 Achieving priority

Priority is an important concept in task scheduling. It can be achieved in several ways. The idea of priority is that some tasks are more important than others, and need to be executed more often and/or have more CPU time.

In the simplest schedulers, tasks have no priority, and simply get serviced as they are reached in the queue (first-come first-served algorithm).

The idea of priority is managed by the operating system by default, but can be influenced by the user or the applications they run. For example, in Linux, tasks can have a priority level running from -20 to 0 and from 0 to +19. Zero is the default priority, larger positive numbers mean a lower priority, and larger negative numbers mean a higher priority. So we might decide to allocate 'MyBirthdayReminder' a priority of 19 because it doesn't need to be serviced very often at all. A video camera recording app may require higher than normal priority, perhaps -5.

In a more complex system, perhaps a server that supports multiple capabilities in a company, priority may be set low for tasks that are not time-sensitive. For example, running a daily stock report does not need to be done in 30 minutes, it can take 6 hours and just as easily complete its purpose on time. Such a task might be assigned low priority so it doesn't reduce the quality of service for users running more interactive or urgent tasks on the server (e.g. processing live customer orders).

What do these numbers actually do? In practice, the priority grading in a particular operating system works in a way that is likely to be specific to that operating system. However, there are two general principles that the scheduler might use:

One option might be to change the length of time allocated to each time-slice given to a particular task. So some tasks might have time-slices of 1ms and others might have 0.9ms, 0.5ms, 0.1ms, and so on. In practice, this may be difficult to achieve.

Another approach is to keep the time-slices uniform, but to skip over tasks in the queue so they get time-slices less often. This is achieved by having a queue in which higher priority tasks can be placed ahead of lower priority tasks.

Another variation is to have several queues, high, low, and medium priority queues, for example, and to manage the issuing of time-slices to threads by placing them in the appropriate queue.

In a specialised operating system, priority models might be managed in different ways from those of a general purpose OS. For example, in an RTOS it is not unusual to have scheduling models that rely upon **Earliest Deadline First** (EDF) priority. This makes sense, since the most urgent task must surely be given priority. In a more general purpose operating system, the idea of explicit deadlines for completion of tasks is less relevant than other considerations.

10.8 Advanced CPU architecture OS support

As the nature of computer operating systems, and their uses, have evolved, so have the demands placed upon processors, and the solutions devised to meet them. If this was not the case, then computer systems would still be using single-core scalar processor architectures.

However, modern processors exhibit a number of specialised capabilities which have only become possible with recent increases in circuit complexity, from millions of transistors per chip in the 1990s, to billions of transistors in the 2020's. The relationship between operating systems and the desire to utilise the extra complexity offered by these new chips has resulted in the processor technologies we see today, and of course they are still evolving.

10.8.1 From scalar to superscalar

As we discovered earlier, in Chapter 3, the limitation of executing one instruction at a time (scalar execution) was quickly recognised as a constraint that could potentially be overcome. The idea stems from the observation that code contains fine-grain parallelism, known as **instruction-level parallelism** (ILP). In other words, neighbouring instructions in a program sequence can often execute independently of each other, whilst some others are dependent upon their neighbouring predecessors, or slightly more distant predecessors. Therefore, a code sequence generally contains a mixture of dependent and non-dependent groups of instructions. As a result, it is possible to opportunistically execute non-dependant instructions simultaneously.

The ability to start multiple instructions simultaneously is known as superscalar instruction issue, and the ability to execute them simultaneously is called superscalar execution. This allows ILP to be exploited as far as is reasonable. However, it was eventually demonstrated that there are

limits to how much ILP can be exploited easily in any given part of a program, even with clever code optimisation algorithms and compilers. Generally, a single program sequence (typically a thread in operating system terms) will rarely be able to execute as many as 4 instructions in parallel for each and every clock cycle. The average over a long program sequence may well be between 2 and 3 instructions per clock (IPC).

Compilers attempt to maximise ILP, and their ability to succeed is certainly influenced by the kind of program being executed. However, there are limits, and the prospects of ILP achieving something like 8 instructions being executed in every clock cycle is unlikely to be obtainable in most general purpose computing contexts.

10.8.2 Multicore

It was eventually realised that investing more and more transistors to deliver higher levels of ILP would not deliver the desired outcome. Extra hardware is costly, and if it delivers a gain of 1% or 2% at best then it may well be a poor design choice. When heat and power are added into the equation, there may even be a case to say that excessive ILP hardware will make performance worse in those terms. Obtaining a 1% performance gain at a cost of a 20% increase in power consumption doesn't sound like a good deal.

For this reason, among others, processor designers began to turn away from ILP as the mainstay of performance increments in next-generation processors. Most current superscalar processors have settled into a 4 to 6 instruction ILP model, but at the same time, processor manufacturers have begun to move toward the idea of using extra available transistors to duplicate CPU cores to run threads in parallel (Thread Level Parallelism or TLP).

Whilst one model was to simply have multiple separate processor chips (multiprocessor system) as we discussed briefly in Chapter 3, this does not exploit the ever-increasing integration possible with VLSI. Designers then began to wonder: why have one over-complex processor on a chip, when we can have 4 more modest and streamlined processors?

This is the basis of multicore processors. The idea of on-chip processing arrays is not new. Indeed, chips have been produced even with thousands of cores for quite a while. However, these tend to be very simple processors. A full-blooded desktop PC processor is a more complex thing. Therefore, most multicore systems currently used in general purpose systems have perhaps 4 to 8 processor cores on a chip.

Because the operating system can be modified to recognise the existence of multiple cores, the task scheduler in particular can run multiple tasks on multiple processors. No longer does the scheduler have to run one thread at a time: it can run 4 or even 8 threads simultaneously with true concurrent operation. This idea can be extended further. In actual fact each of the cores can run a scheduling queue independently, so that we may have a number of threads being multi-tasked on each core. The workload can then be spread across the cores to maximise performance.

Obviously this boosts performance considerably in theory. But remember that a single chip with 8 processor cores still has one system bus, one physical memory and one set of IO resources, and that these must be shared among the 8 cores.

This memory access bottleneck can be augmented by using suitable cache design. If each core has its own instruction and data cache, and the whole chip also has a unified cache to tie together these eight cores to memory, then the pressure on the memory bus could be reduced very significantly.

Therefore, a typical multicore architecture will have a structure that includes multi-level cache hierarchy, methods of dealing with memory coherence, which we encountered in earlier sections, and some capability to exchange information rapidly between cores.

10.8.3 Hyperthreading and SMT

Given that a multicore processor allows an operating system to run a separate scheduling queue on each core, there is an initial boost in performance that could be expected to be up to 'n' times more performance when 'n' cores are used.[135] However, the act of scheduling and switching of threads on those individual cores is still no more efficient than it is on a single core.

[135] In practical terms the gain will never be quite as much as n, due to overheads in the operating system, even if all of the other resources are in unlimited supply. Amdahl's Law once again!

Remember the example in Chapter 8, where a reader tries to read five books at once, and the granularity of their reading is reduced incrementally. Reading one chapter of each book in turn, then one page, and then one paragraph, leads to less and less reading being done, and more and more time being spent swapping between books, putting in bookmarks, etc.

This principle nicely illustrates the problem with task scheduling and task switching. This is partly due to the on-chip **state** of a processor core. Processor state relates to things like the contents of registers, buffers,

the execution pipeline hardware, and other items. Switching all of that state information when a task switch occurs is costly, taking tens or hundreds of CPU clock cycles per task switch.

We also know that when a thread hits a limit in terms of its instruction level parallelism, it needs to wait until the processor can clear the backlog of waiting instructions. Fine-grain parallelism in a single thread has unavoidable limitations.

Hyperthreading/SMT[136] is a technique that attempts to optimise these problems. The idea is that instead of having a single set of registers, cache, etc, for each core, and then swapping that state content to and from memory in a coarse fashion, when a task switch occurs, it is possible to duplicate the state related circuits of the core for several threads. So there may be perhaps 2 or 4 thread states held actively in hardware, but still only one set of computational hardware for computation, memory access, cache, etc.

[136] In practice, Hyperthreading is the same conceptually as SMT (Simultaneous Multithreading). Strictly speaking, Hyperthreading is an INTEL branded SMT mechanism, but the term has caught on in general use as a byword for SMT.

As a result of this approach, multiple threads can coexist, and each thread can attempt to issue instructions at the same time. They will of course compete over the limited resources available for computation, such as integer and floating point ALU's, and memory ports. But this is potentially quite a good thing as it ensures all of these resources are heavily utilised.[137] The result is that each thread will be able to progress small pockets of instructions at a time, intermixed with the work of other threads.

[137] Maximum performance occurs when all CPU resources are used all of the time, known as peak utilisation.

An example of this is shown in Figure 10.3. Here, if a particular thread gets **stalled**, because it is waiting for memory for instance, then the other threads can use those resources more than normal, and make more progress. In this way, threads can exploit parallelism not just via ILP within an isolated thread but in terms of a fine level of granularity across threads.

Without hyperthreading/SMT, the stalled thread would just wait, wasting CPU time, because only one thread is active at a time. We can see in Figure 10.3 that the SMT approach allows the same work to be completed more quickly: in other words, the performance of the system has been increased significantly, as has the utilisation of CPU resources.

As before, the fact that 4 threads can coexist will not necessarily deliver 4 times the performance. But if 4 threads can deliver 2 times the performance that is still a big gain. Ultimately, in a system with 8 cores and 4-way hyperthreading, the best possible performance gain might be

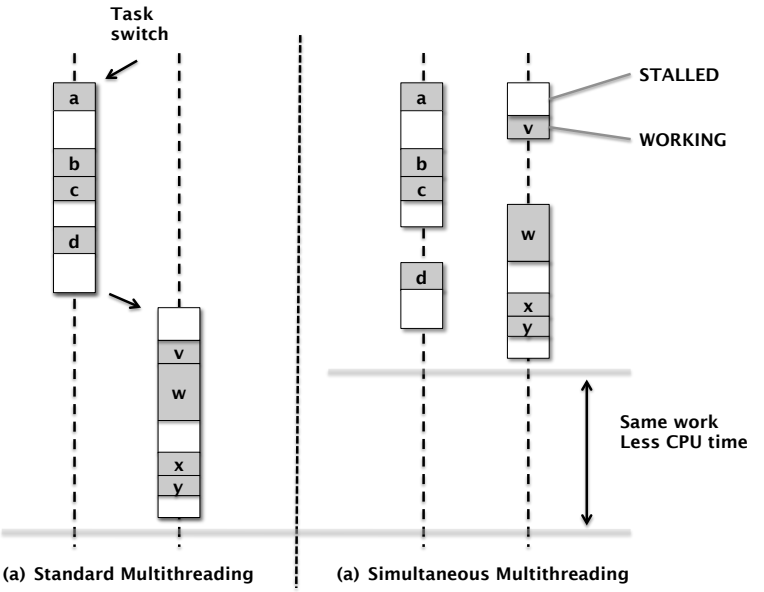


Figure 10.3: SMT versus traditional multithreading. (a) shows two independent threads being switched back and forth via traditional multithreading. (b) shows two threads operating as simultaneous multi-threads, both attempting to progress in a piecemeal fashion in competition with the other.

Table 10.1: Examples of hyperthreaded multicore processors. Benchmark scores via cpu.userbenchmark.com (64-core-point test).

	Price £	Cores	Threads	Benchmark
AMD Ryzen TR 2920X	388	12	24	1925
AMD Ryzen TR 2990WX	1490	32	64	4142
INTEL Core i7-9700KF	360	8	8	1062
INTEL Core i9-9980XE	1850	18	36	2851

imagined to be 32 times the performance. But this would never be achievable in practice. However, a real performance gain of say 8 could well be achievable with the right workload.

It is important to note that whilst a core might only support 4 hyperthreads, these can still also be individually switched via task switching, as well as running simultaneously with others. In practice, a hyperthread may actually be a group of alternating threads running in parallel with another group of hyperthreads alongside and doing the same thing.

Examples of the performance impact of cores and hyperthreading support are given in Table 10.1, to illustrate this point. For the particular benchmark used (of which there are many), we can see that a processor with nearly three times as many cores and threads (AMD 2990WX vs 2920X) achieves about 2 times the performance. Meanwhile a processor with around double the number of cores but four times the threads (INTEL 9700KF vs 9980XE) gains almost three times the performance.[138]

[138] We should be cautious here of course, one benchmark may reflect particular capabilities and not a general level of performance. We should always choose benchmarks carefully to reflect the intended use of the component.

An important lesson here, therefore, is that increasing resources doesn't automatically translate into equal increases in performance; the situation is far more subtle.

10.9 Summary

Exploring the nature of operating systems, from the workload management perspective, we should have been able to see ultimately that a modern computer system is a cleverly coordinated collection of software elements. Rather than a few desktop packages running in their respective windows, the reality is that we can have literally hundreds of individual software parts running side by side, each playing an essential part in some aspect of managing the smooth running of the system, and your applications.

Not only do many software elements have to operate together in the same workspace, but they also have to communicate information to each other, or preserve their information privacy. This is achieved via the operating system too. Indeed, some of these aspects are essential for maintaining security and reliability in computer systems.

Finally, we learned that the illusion of multi-tasking is often just that. Even though we perceive many processes, applications and threads running in parallel, it is often the case that these components are actually being sliced up into small parts, and serviced piece by piece, in an interleaved fashion. The time-slicing, and the scheduling of these actions, lies at the heart of the modern operating system.

Looking toward the future we have observed how the current generation of processor architectures have begun to evolve toward supporting these demanding requirements. The emergence of powerful hyperthreaded architectures and their continued development will undoubtedly shape the next ten years of computer systems architecture.

10.10 Terminology introduced in this chapter

Context switch	Deterministic
FIFO	Hyperthreading
ILP	Instruction-Level parallelism
IO Wait Queue (task)	IO-Polling
Jitter (task)	Kernel level thread
PCB	PID
Prioritisation	Priority level
Process control block	Process identifier
Processor state	Read Queue (task)
Repetitive Interrupt Timer	Resumed task
RIT	Scheduler
Scheduling algorithm	Simultaneous Multithreading
SMT	Stalled thread
Suspended task	Task list
Task priority	Task queue
Task scheduler	Task-switching overhead
User level thread	Yield control (task)

These terms are defined in the glossary, Appendix A.1.