Lecture 3

Gidon Rosalki

2025-04-06

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

1 Processes

The process for an OS running programs is as follows: You take your code (for example in C), compile it into a .out file, which has a data segment, and an instructions segment. From here these are both loaded into memory. The memory has the stack which grows downwards towards the heap, the heap that grows upwards towards the stack, the data is stored under the heap, and the instructions / text under the data, and is then run on the CPU. So the CPU sits between both the instruction memory, and the data memory.
Every iteration the execution is as follows: Fetch the instruction at PC, and save it in the instruction register (IR). Decode the instruction in IR. Execute (possibly using some registers). Write the results to the registers / memory. Move the PC on to the next instruction, and then repeat. Let us consider the instruction `x = x + y`. We shall suppose that x is stored at 100, and y is stored at 104. Our required asm instructions are:

```
lw r1, 100
lw r2, 104
add r3, r1, r2
sw r3, 100
```

One may follow the execution steps above for the given asm listing. The execution contexts are as follows: The CPU stores the values in the registers. These include the PC, standard registers, and other "special registers" such as the stack pointer (SP), which points to the root of the data. The memory stores what's written in the memory, such as the code/text, data, stack, heap, and some internal data in the OS relevant to the program such as the user, priority, and so on.

A **process** is an instance of a program execution. It may be considered an abstraction of "an individual computer". The processes are OS objects, they provide context of execution, and the process exists only for the duration of the execution. These processes are also called **jobs**. A process is executing on a CPU when it is resident in the CPU registers. For non-resident processes, the context part that was "in the CPU" is stored in a special location in memory.
So what's the difference between a process and a program? A **program** is a passive entity, and a process is active. A program becomes a process when an executable file is loaded into memory. This is done by the **loader**, which is a component of the OS. There is more to a process than just a program, a program is just part of the process state, and there is less to a process than a program, since a program can invoke more than one process.
A process needs an address space, so here is a simplified overview. The address space is a set of accessible memory addresses by the process. This will only be part of the memory, not all of it. We may consider that this space is defined by 2 values, the base and the bound. So the process can only access addresses in $[\text{base}, \text{base} + \text{bound}]$. This is verified by the CPU in hardware. Protection / isolation ensures that processes cannot access each others memory. Address translation is also provided, this is when a process specifies address x, it actually means $\text{base} + x$. So this way the same pointer address in different processes points to different points in memory. The OS address space is called **kernel memory**, and the rest is **user memory**..
So what do we need to save when switching between processes 1 and 1? We need to save the context, and copy everything in the CPU. We need not save the memory, since that is in that process' address space, and thus it will remain unaffected. So we'd need to save the PC, but not the code, and the SP, but not the entire stack.
A process is represented in the OS through a Process Control Block. This contains

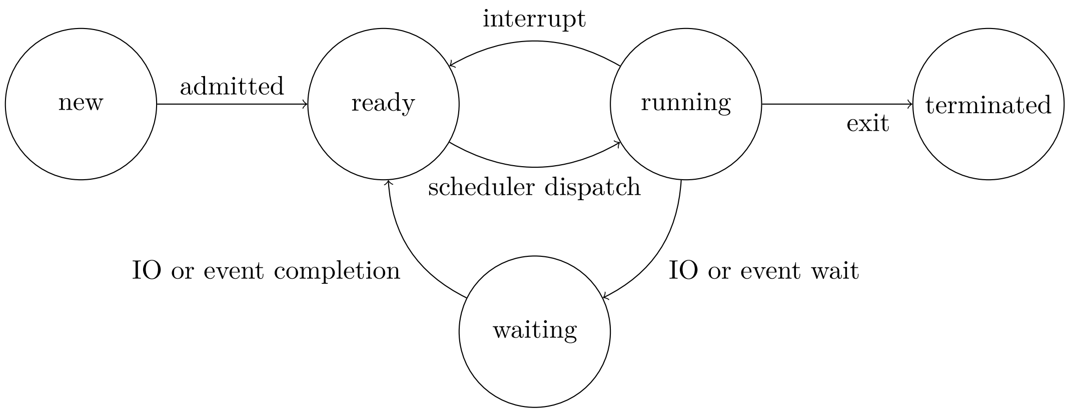
- Process ID
- Process state
- User - Needed for protection, what is the process allowed to do?
- Accounting information

- Priority - needed for scheduling decisions, when should the process run?
- Allocated memory - Point to relevant data in other OS data structures.
- Open files
- Open communication channels
- Storage space for the CPU state (registers) - for when the process is not being run, and from here it will be restored when it is run again.

It should be noted that this is a naive overview, which will be discussed in further depth later on in the course.

A process will change state over its life cycle:

- new: The process is being created
- running: Instructions are being executed
- waiting/blocked: The process is waiting for some event to occur
- ready: The process is waiting to be assigned to a processor
- terminated: The process has finished execution.



1.1 Scheduling

Which process should get to run on the CPU, and for how long? There are many algorithms and methods for choosing. We can for example just choose the process that has not run for the longest time. Occasionally schedulers and dispatchers are separated. The Scheduler decides which process should run, and for how long. The dispatcher is the module responsible for executing the CPU scheduler decisions, such as context switching, saving state, and so on.

To recap, when we want to context switch between processes, we must

- Save the processor context (including PC, and other registers)
- Update the PCB with the new state and accounting information
- Move the PCB to the appropriate queue (ready, blocked/waiting, etc)
- Update the PCB of the selected process
- Update memory-management data structures
- Restore the context of the selected process

Remember, this is a naive overview, which will change to be more in depth in the future.

Let us consider creating processes using `fork`. `fork` is the system call that creates processes. The child and parent results from fork are **identical** (aside from the process number, which is unique).

Sometimes processes want to cooperate with each other, for example sharing information, breaking a large task into numerous smaller tasks, and so on. On the other hand, processes should be protected from each other. So we use Inter Process Communication (IPC), which is provided by the OS, through very specific system calls, but be warned, this comes at a very high overhead. There are many ways to achieve IPC. We can specify a segment in the memory that is shared, and all processes can access it. This is often done through files, one writes to the file, and the other reads from it. We can also use the socket interface to exchange messages through a communication channel. Sometimes this is classified into shared memory, or message passing systems. This overall creates many synchronisation problems.

2 Threading

A **thread** is a path of execution. Until now, each process has had one thread of control, which was defined by the PC and the stack. In multithreading, we have multiple independent execution paths. Though there is a shared context (memory contents, open files, and so on). Many applications need to perform more than one task at once. Consider the web browser, it needs to both retrieve data from the network (slow), and display images of data. Also the server, which needs to handle many clients at once, where handling each client is again a slow process. Threading allows concurrency in performing tasks. Without concurrency a word processor would wait on each keystroke, evicting the CPU, and no other tasks in the word processor would run until the keystroke has interrupted the CPU again. If each task is a process then it must use IPC liberally, which is very slow. However, if we use a multi threaded process, each thread has its own registers (including PC and stack), and all the threads share the program code, data, and files.
A process virtualises the **computer**, each running application sees an abstract computer dedicated to itself. However, a thread virtualises the **core**, an application can be structured as if it will run on multiple cores, regardless of how many are actually available. Concurrency - threads share a single physical core (time slicing), and true parallelism is when threads run on distinct processors.
The kernel maintains context information for the process and the threads. Blocking a thread does not have to block the process. The disadvantage of threading is that switching between threads requires the kernel. **Kernel level threads** are managed by the kernel, but run in **user** space. **Kernel threads** are managed by the kernel, and run in **kernel** space. These are used for structuring the OS. Each process has at least one thread The CPU state is actually stored per thread. Scheduling is performed per thread, and the **Thread Control Block (TCB)** points to PCB (at least conceptually).
Kernel level threads have proven more flexible than processes, but we still need to muck around a lot with the scheduler, and system calls. In **user level threads**, all thread management is done by the user application / library (in user mode). The kernel is **not aware** of the existence of threads. Thread switching additionally does not need kernel mode privileges. Scheduling is application specific. However, This has the major problems that system calls by threads (including I/O), will **block** the whole process. A single thread can monopolise the time slice, thus starving the other threads within the task.

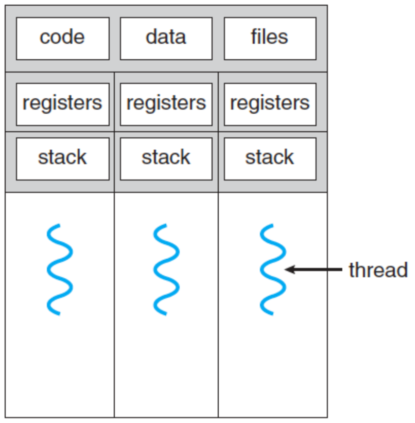


Figure 1: Multi threaded processes

Benefits of threads: It's around 30-100 times faster to create a new thread than a process. Termination is also faster. It is also about 5 times faster to switch between 2 threads within the same process, than to switch processes. Threads within the same process share memory and files, and thus they can communicate without invoking the kernel/IPC, and it is thus much faster.

Processes	Kernel level threads	User level threads
Protected from each other, require the OS to communicate	Share address space, simple communication, useful for application structuring	
High overhead, all operations require a high cost kernel trap	Medium overhead, operations require a low cost kernel trap	Low overhead, everything is done at the user level
Independent, if one blocks this does not affect the others		If a thread blocks, the whole process is blocked
Can run in parallel on different processors in a multiprocessor		All share the same processor, so only one runs at a time
System specific API, programs are not portable		The same thread library may be available on several systems
One size fits all		Application specific management is possible

Table 1: Threads vs processes summary

IS the CPU "aware" of the existence of

- User level threads
- Kernel level threads
- Both user and kernel level threads
- Neither user nor kernel level threads

The CPU is unaware of all these, they happen at the OS level.

2.0.1 Hardware level threads / hyperthreading

Intel's technology since the pentium 4 (2002). Each core, even though it is physically a single core, presents as 2 to the OS. The management (swithcing between hyper threads) is handled by the CPU in hardware, and it is more efficient than kernel level threads. Currently each core is split into 2 hyperthreads, so a 4 core machine will appear to have 8 cores to the OS.