

Lecture 5: Feb 2

*Lecturer: Prashant Shenoy**TA: Vimal Mathew & Tim Wood*

5.1 Interprocess Communication

Inter-Process Communication (IPC) is a set of techniques for the exchange of data among two or more threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for message passing, synchronization, shared memory, and remote procedure calls (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated.

5.1.1 Producer and Consumer

The producer and consumer problem is one where two processes must coordinate to exchange data. In this system, a *producer* process is periodically creating new data elements and *consumer* process is waiting for these data items to be created and is using them for some other task. In order for this system to function, the producer and consumer require a communication process to allow them to coordinate when the producer has created a new item so that the consumer can successfully read the data. Such a system can be built using either message passing or a shared memory approach.

5.1.2 Message Passing

Message passing is a form of communication used in interprocess communication. Communication is made by the sending of messages to recipients. Each process should be able to name the other processes. The producer typically uses *send()* system call to send messages, and the consumer uses *receive()* system call to receive messages. These system calls can be either synchronous or asynchronous, and could either be between processes running on a single machine, or could be done over the network to coordinate machines in a distributed system. This allows the producer to transfer data to the consumer as it is created.

5.1.3 Shared Memory

Shared Memory is an OS provided abstraction which allows a memory region to be simultaneously accessed by multiple programs with an intent to provide communication among them. One process will create an area in RAM which other processes can access (this is typically done using system calls *mmap*, *shmget* etc). Normally the OS prevents processes from accessing the memory of another process, but the Shared Memory features in the OS can allow data to be shared. Since both processes can access the shared memory area like regular working memory, this is a very fast way of communication (as opposed to other mechanisms of IPC). On the other hand, it is less powerful, as for example the communicating processes must be running on the same machine (whereas other IPC methods can use a computer network), and care must be taken to avoid issues if processes sharing memory are running simultaneously and may try to edit the shared buffer at the same time.

5.2 Threads

A thread is a sequential execution stream within a process. This means that a single process may be broken up into multiple threads. Each thread has its own Program Counter, registers, and stack, but they all share the same address space within the process. The primary benefit of such an approach is that a process can be split up into many threads of control, allowing for *concurrency* since some parts of the process to make progress while others are busy. Threads can also be used to modularize a process – a process like a web browser may create one thread for each browsing tab, or create threads to run external plugins.

5.2.1 Processes vs threads

One might argue that in general processes are more flexible than threads. For example, processes are controlled independently by the OS, meaning that if one crashes it will not affect other processes. However, processes require explicit communication using either message passing or shared memory which may add overhead since it requires support from the OS kernel. Using threads within a process allows them all to share the same address space, simplifying communication between threads. However, threads have their own problems: because they communicate through shared memory they must run on same machine and care must be taken to create thread-safe code that functions correctly despite multiple threads acting on the same set of shared data. Additionally, since all threads are within a single process, if one thread has a bug it can corrupt the address space of all other threads in the process.

When comparing processes and threads, we can also analyze the context switch cost. Whenever it is needed to switch between two processes, we must invalidate the TLB cache which can be a slow operation. When we switch between two threads, on the other hand, it is not needed to invalidate the TLB because all threads share the same address space, and thus have the same contents in the cache. Thus the cost of switching between threads is **much** smaller than the cost of switching between processes.

5.2.2 Kernel Threads and User-Level Threads

Threads can either be created as kernel threads or user-level threads depending on the operating system. In systems that use **kernel-level threads**, the OS itself is aware of each individual thread. A context switch between kernel threads belonging to the same process requires only the registers, program counter, and stack to be changed; the overall memory management information does not need to be switched since both of the threads share the same address space. Thus context switching between two kernel threads is slightly faster than switching between two processes. However, kernel threads can still be somewhat expensive because system calls are required to switch between threads. Also, since the OS is responsible for scheduling the threads, the application does not have any control over how its threads are managed.

A **user-level thread** is a thread within a process which the OS does *not* know about. In a user-level thread approach the cost of a context switch between threads can be made even lower since the OS itself does not need to be involved—no extra system calls are required. A user-level thread is represented by a program counter, registers, stack, and small thread control block (TCB). Programmers typically used a *thread library* to simplify management of threads within a process. Creating a new thread, switching between threads, and synchronizing threads are done via function calls into the library. This provides an interface for creating and stopping threads, as well as control over how they are scheduled.

When using user-level threads, the OS only schedules processes, which in turn are responsible for scheduling their individual threads. Unfortunately, since the threads are *invisible* to the OS, the OS can make poor decisions such as scheduling a process with idle threads or giving unbalanced CPU shares between processes that have different numbers of threads. Perhaps the greatest disadvantage of user-level threads is that if

a single thread performs an I/O request, the OS scheduler may cause the entire process, and all its other user-level threads, wait until the I/O finishes before returning the process to the run queue. This can prevent applications with user-level threads from achieving high degrees of concurrency if threads are performing I/O. Solving these problems requires communication between the kernel and the user-level thread manager. Another limitation of user level threads is that they cannot be used to spread tasks across multiple cores in modern CPUs. This is because a process is only scheduled to run on a single CPU core at a time.

5.2.3 Threading Models

Some operating systems such as Solaris support the concept of a *Lightweight Process*. This is a hybrid approach where the OS is able to take a thread and map it to one or more lightweight processes. Each lightweight process can be associated with multiple threads, and each thread can be associated with multiple processes. This means that if one thread needs to block for I/O, the lightweight process it is associated with will block, but the other threads hooked to that process may be able to continue running within a different lightweight process.

Approaches such as this allow for a flexible mapping between threads and processes. When using user-level threads, the system is using a *many-to-one* model since many threads are allocated to each process. Kernel level threads use a *one-to-one* model since each thread is given a process. The Solaris approach of Lightweight Processes is a two-level threading model where threads can be mapped to either its own process or to several processes.

5.2.4 Threading Libraries

Thread libraries provide programmers with an API for creating and managing threads. The thread library can be implemented either completely in user space, or it can be supported by the OS kernel itself.

5.2.4.1 Pthreads

Pthreads is a threading API following the POSIX standard. Pthreads is a library for C programs and is supported by a variety of operating systems. When a programmer designs an application to use the Pthreads API, then it can generally be ported between different systems without requiring the source code to be modified.

5.2.4.2 Java Threads

The Java programming language also provides support for threads directly (as opposed to through an additional library such as Pthreads). The Java Virtual Machine uses different types of threads depending on the OS in which it is run. If the host OS does not support kernel threads, then the JVM will use a built-in user level threads package, otherwise it will use kernel-level threads. In either case, the code used for the java program is the same; the JVM transparently maps the programmers threads into either kernel or user-level threads. Since most modern operating systems now support kernel-level threads, the JVM will typically use kernel threads.

5.3 Scheduling

Scheduling is a key concept in computer multitasking and multiprocessing operating systems. It refers to the way processes are selected to be run, and how they are allocated time on the CPU. The operating system can support multiple scheduling policies which impact how processes or threads are chosen to be run and in turn impact the performance seen by applications.

5.3.1 Types of Schedulers

Operating systems often utilize two types of schedulers: a long-term scheduler (also known as an admission scheduler or high-level) and a short-term scheduler (also known as a dispatcher). The names suggest the relative frequency with which these functions are performed.

5.3.1.1 Long-term Scheduler

The long-term, or admission, scheduler decides which jobs or processes are to be admitted to the ready queue and how many processes should be admitted into the ready queue. This controls the total number of jobs which can be running within the system. In practice, this limit is generally not reached, but if a process attempts to fork off a large number of processes it will eventually reach an OS defined limit where it will prevent any further processes from being created. This type of scheduling is very important for a real-time operating system, as the system's ability to meet process deadlines may be compromised by the slowdowns and contention resulting from the admission of more processes than the system can safely handle.

5.3.1.2 Short-term Scheduler

The short-term scheduler (also known as the dispatcher) decides which of the ready, in-memory processes are to be executed (allocated a CPU) next following a clock interrupt, an IO interrupt, or an operating system call. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term schedulers - a scheduling decision will at a minimum have to be made after every time slice, which can be as often as every few milliseconds. This scheduler can be *preemptive*, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or *non-preemptive*, in which case the scheduler is unable to "force" processes off the CPU.