Lecture Notes for CS347: Operating Systems

Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay

2. Processes and Threads

2.1 Life cycle of a process

- Recall that the process is a basic unit of execution in an OS. The main job of any OS is to run processes, while managing their lifecycle from creation to termination. Processes are typically created in Unix-like systems by forking from an existing process. The OS starts the first `init` process after bootup, and all subsequent processes are descendents of this process.

- A process can create a child process using the `fork` system call. After the fork, the memory image of the child process is a complete copy of the memory image of the parent (the child and parent memory images may diverge subsequently). The fork system call returns in both the parent and child processes, with different return values. In the parent, it returns the `pid` of the new child process. In the child, it returns 0. Both processes then resume execution from the instruction right after fork, and can be scheduled as independent entities by the CPU scheduler.

- A typical use case of fork is to create a child and run the `exec` system call in the child. The `exec` system call loads a new executable into the memory image of the process calling it, enabling the child process to do something different from what the parent is already doing. All processes (beyond the first process) are created with this fork+exec combination in Unix-like operating systems.

- A simple OS would create a complete copy of the parent's memory image for the child. However, since most children rewrite the parent's memory image with their own anyways, most modern operating systems follow a technique called "copy-on-write" (CoW) during fork. With CoW, no copy of the parent's image is made during fork. Instead, both parent and child receive the same read-only copy of the parent's image. When either wishes to modify this image, a trap is generated, and the OS then proceeds to copy and give separate memory images to the parent and child while servicing the trap.

- A process can terminate itself using the `exit` system call. Alternately, a process can be terminated by the OS under several circumstances (e.g., when the process misbehaves). Once a process gives up the CPU for the last time after the exit system call, the OS can proceed to reclaim the memory of the process (note that the memory of a process cannot be reclaimed *during* the `exit` system call itself, while the CPU is executing on its memory). While there are several places in an OS where the memory can be reclaimed, most operating systems leave it to the parent to reclaim a dead child's memory. This serves the additional purpose of returning values from the child to parent as well.

- Every process that terminates must be *reaped* by its parent using the `wait` system call. When the parent calls `wait`, the return value of the child's termination is returned to the parent. The

state/memory of a dead process is cleared only after it is reaped by its parent. Therefore, after a child process terminates, it exists as a **zombie** until the parent process calls `wait` and reaps it. When the parent process itself terminates before the children, the orphan children are adopted and reaped by the `init` process. (Some operating systems may also terminate all children upon termination of parent.)

- The `wait` system call has several variants provided by the C library. When a parent calls the `wait` function of the C library, the OS will return the exit value of any of its dead children. Alternately, the parent can used `waitpid` to wait for a specific child by providing the `pid` of the child as an argument to the system call. The `wait` system call is generally blocking, though it can be invoked in a non-blocking fashion by providing suitable options during invocation.

- The command shell provides a good case study on the lifecycle of processes. Upon receiving a user's input, the shell forks a new child, and executes the command in the child. For foreground commands, the shell waits for the child process(es) to complete before asking for the next input. For background processes, the shell reaps terminated children at a later time, and asks the user for the next input without waiting for the background command to complete.

- In summary, every process exists in one of the following states (there may be slight differences across operating systems):

  1. **New.** A process that is in the process of being created, and has never executed so far.
  2. **Running.** A process that is currently executing on a CPU.
  3. **Blocked / waiting.** A process that is blocked on some event, and cannot execute.
  4. **Ready.** A process that is ready to execute and is waiting to be scheduled.
  5. **Terminated / zombie.** A process that has exited and is waiting to be reaped by its parent.

2.2 Process Control Block

- All state pertaining to a process is stored in a kernel data structure, generally referred to as the **process control block (PCB)**. Note that the exact name of the data structure may differ from OS to OS. For example, it is called `task_struct` in Linux. The kernel maintains PCBs in a linked list or some other suitable data structure within its memory space. Some important information stored in the PCB includes:

  1. The process identifier. Every executing process typically has a unique identifier in every OS. For example, this identifier is referred to as `pid` in Linux.
  2. The identifier and pointers to its parent and child processes.
  3. The current state of the process (running / blocked etc.).
  4. Pointer to the kernel stack of the process for execution in kernel mode. The PCB may also hold pointers to various data structures on the kernel stack, such as the process context, consisting of the values in its program counter and other CPU registers. This information is not updated in the PCB every time the registers change, but is only stored in the PCB when the process context has to be saved, e.g., when moving from user mode to kernel mode, or during a context switch.
  5. Information related to scheduling the process (e.g., its priority for scheduling).
  6. Information related to the memory management of the process (e.g., pointers to page tables).
  7. Information about list of open files, files descriptors, and other information pertaining to I/O activity.

- Note that the PCB of a process is the "brain" of the process, and contains all the critical information required for a process to execute on a system. The PCB is created when a process is born via `fork`, and is reclaimed when a process is terminated. While systems calls such as `exec` rewrite the memory image of the process, the PCB (and the entities pointed by it, like the kernel stack) largely remain intact during `exec` or any other system call, except for slight modifications (like changing the page tables to point to the new memory image).

- *A note on file descriptors.* We will study file descriptors in detail later on, but here is a short introduction. Every process maintains an array to keep track of its open files/sockets/pipes as part of its PCB, and file descriptors serve as an index into this array. By default, file descriptor number 0 points to standard input, 1 to standard output, and 2 to standard error. Other files/sockets/pipes opened by the process will get the first available free index in order. A process gets a file descriptor from a kernel when it opens a file/socket/pipe. All subsequent operations on the file/socket/pipe, such as the `read` or `write` system calls, will reference this file descriptor.

  The fork system call duplicates all the file descriptors of the parent into the child, so that the file descriptors in the parent and child point to the same file. Most operating systems also provide the `dup` system call, which duplicates a file descriptor to create a copy that points to the same file as the original file descriptor.

Note that the fact that the `exec` system call replaces the memory image of the process but keeps its array of file descriptors intact is made use of cleverly by most shells. Pipes and I/O redirection are implemented in the shell by clever manipulations of the file descriptor table. For example, to implement output redirection from a command, the shell forks a child process to implement the command, then opens the file to which the output must be redirected. It then manipulates the child's standard output file descriptor to point to the new output file (by closing the original standard output file descriptor of the child, and duplicating the new output file descriptor it wishes to send output to into the standard output slot of the array). Subsequently, the child does `exec` to execute the command, but its output file descriptor points to the output file setup by the parent, and not to the console. So, while the child process assumes it is writing to the standard output console, it is in fact writing to a file. This is one of the reasons why Unix-like systems use different system calls like `fork` and `exec` instead of one system call to spawn a child with a new executable: the separate system calls let the parent manipulate the child's file descriptors before it goes off to run its own executable.

2.3 Inter-process communication (IPC)

- Processes often need to communicate with each other in order to accomplish useful tasks. The OS provides several mechanisms to enable processes to communicate with each other.

- **Shared memory** is the simplest way in which two processes can communicate with each other. By default, two separate processes have two separate memory images, and do not share any memory. (Note: a forked child has the same memory image as the parent at creation, but any changes made to the child's memory will not be reflected in the parent.) Processes wishing to share memory can request the kernel for a shared memory segment. Once they get access to the shared memory segment, it can be used as regular memory, with the property that changes made by one process will be visible to the other and vice versa.

  A significant issue with using shared memory is the problem of synchronization and race conditions. For example, if several processes share a memory segment, it is possible for two processes to make a concurrent update, resulting in a wrong value. Therefore, appropriate synchronization mechanisms like locking should be used when using shared memory.

- **Signals** are another light-weight way for processes and kernel to communicate with each other, and are mainly used to notify processes of events. For example, when a user hits Ctrl+C, the OS sends a a signal called SIGINT to the running process (as part of handling the interrupt generated by the keyboard). When a process receives a signal, the normal execution of the process is halted, and a separate piece of code called the signal handler is executed by the process. A process template comes with default signal handlers for all signals, so that every program will not need to have code to handle signals. For example, for the Ctrl+C signal, the process will terminate by default. However, a program can have its own in-built signal handling function, that can be passed to the kernel with the `signal` system call. The OS will then invoke this new function when the process receives a signal. Signals are not just for communication between the OS and processes. One process can use signals to communicate with another process also. The `kill` system call can be used by a process to send a signal to another process whose `pid` is specified as an argument to the system call.

- **Sockets** are a mechanism to communicate between two processes on the same machine, and even between processes on different machines. Network sockets are a standard way for two application layer processes running on different machines (e.g., a web client and a web server) to exchange data with each other. Similarly, two processes on the same machine can use Unix domain sockets to send and receive messages between each other. The usage of Unix domain sockets is very similar to the usage of network sockets. That said, sockets are more widely used for communicating across hosts than across processes on the same host.

  Sockets present an interesting case study on blocking vs. non-blocking system calls. Some socket system calls (e.g., `accept`, `read`) are blocking. For example, when a process reads from a socket, the system call blocks until data appears on the socket. As a result, while the process is blocked, it cannot handle data coming on any other socket. This limits the number of concurrent communications a process can sustain. There are several techniques to fix this problem. A process could fork off a new child process for every connection it has, so that a

child process can be dedicated to reading and writing on one connection only. Alternately, a socket can be set to be non-blocking, and the process can periodically poll the socket to see if data has arrived. Finally, system calls such as `select` can be used to get notifications from the kernel on when a socket is ready for a read.

- **Pipes**. A pipe is a half-duplex connection between two file descriptors — data written into one file descriptor can be read out through the other. A pair of file descriptors can be bound this way using the `pipe` system call. The two ends of the pipe are referred to as a read end and a write end. Reading from and writing to a pipe can be blocking or non-blocking, depending on how the pipe is configured. The data written to a pipe is stored in temporary memory by the OS, and is made available to the process that reads from it.

  Pipes are anonymous, i.e., there is no way to refer to them outside the process. The typical use case of pipes is for a parent process to create a pipe, and hand over endpoints to one or more children. Alternately, named pipes or FIFOs (First-In-First-Out) enable a process to create a pipe with a specified name, so that the endpoints of the pipe can be accessed outside the process as well.

- **Message passing** is another IPC mechanism provided by many operating systems. A process can create a message queue (much like a mailbox), and another process can send a message to this queue via the OS. A message queue is maintained as a linked list inside the kernel. Every message has a type, content, and other optional features like priority. System calls are available to create a message queue, and post and retrieve messages from the queue. As in the previous cases, blocking and non-blocking versions of the system calls are available. For example, when the message queue is full, the writing process can choose to block or return with an error message. Similarly, when the message queue is empty, the system call to read a message can block or return empty.

2.4 Threads

- Multiprogramming or concurrency is the concept of having multiple executing entities (e.g., processes) to effectively utilize system resources. For example, if a system has multiple processes, then the CPU can switch to another process when one process makes a blocking system call. Multiprogramming is also essential to effectively utilize multiple computing cores that are common in modern systems.

- An application (say, a web server or a file server) that wishes to achieve efficiency via multiprogramming can fork multiple processes and distribute its work amongst them. However, having a lot of processes is not efficient because every process consumes memory, and sharing information across processes is not easy. Threads are a type of light-weight processes that are widely used in such situations.

- Every process has a single thread (of execution) by default, but can create several new threads once it starts. Threads of a process share memory corresponding to the code and data of the process, which makes it easy to share data between them. This is also the reason why creating threads consumes fewer resources than creating processes. Each thread represents a separate unit of execution, and one thread of a process can execute while the other blocks. Because threads can be executing on different parts of the process code, each thread has its own program counter and CPU register values. Threads can optionally have some thread-specific data.

- POSIX provides an API for creating and managing threads. Linux implements these POSIX threads or `pthreads`. When a process creates a thread using `pthread_create`, it provides a pointer to a function that the thread must run. A thread completes execution when the function exits. The parent process can choose to wait for all threads it created to complete (this is called a join operation), or it can detach the threads and not wait for their completion.

- Sharing data across threads is easy, as they share all data in the program. However, race conditions can occur when two threads concurrently update a shared variable. So care must be taken in designing multi-threaded applications. When updating shared data structures, threads must either access mutually exclusive portions of the data, or must lock and coordinate when accessing the same data.

- Most real-life applications are multi-threaded. Some applications spawn a fixed number of threads, and distribute the work of the application amongst them. For example, a web server can have a fixed number of "worker" threads. When a request arrives from a client, a free thread picks up the request and serves it. Having multiple threads also means that one thread can block while serving a request, without impacting the server's ability to serve another request.

- So far, our discussions of threads have referred to the threads that an application program creates. These are called **user threads**. Do these user threads always map to separate units of execution at the kernel? That is, are all the threads in an application scheduled as independent entities by the CPU scheduler? The answer is dependent on the architecture of the particular OS. The separate threads of execution that the kernel deals with (e.g., for the purpose of

scheduling) are called **kernel threads**. In some operating systems (e.g., Linux), there is a one-to-one mapping between user threads and kernel threads. That is, when an application program creates a thread using the `pthreads` library, Linux creates a new kernel object corresponding to the thread. (In fact, Linux creates processes and threads much the same way, with the only difference being the amount of information that is cloned during fork.) Thus, there is a one-to-one mapping between the POSIX threads created by an application programs and kernel threads managed by Linux. (Note that Linux also has some kernel threads that do not correspond to any user threads, but whose sole purpose is to carry out kernel tasks.)

- However, there is not always a one-on-one mapping between user and kernel threads. In some thread libraries on some operating systems, multiple user threads are multiplexed onto the same kernel thread (i.e., many-to-one mapping). While the user program sees multiple threads, the kernel only perceives one thread of execution. So, when one of the user threads blocks, the other user threads that share the same kernel thread will also be blocked. A user-level library takes care of scheduling the multiple user threads onto a single kernel thread. Such user-level threads do not achieve all the benefits of multiprogramming. However, user threads are easier to create and manage, and enable the application to handle and switch between many concurrent operations at a time.

- Finally, there is also a middle ground between one-to-one and many-to-one mappings, which is a many-to-many model. In some operating systems, the kernel multiplexes several user threads onto a smaller number of kernel threads.

- Operations such as forking or handling signals becomes tricky in a multi-threaded application. When forking a child, some operating systems fork all the threads in the process, while some only fork the thread that made the system call. When handling signals, a multi-threaded application can designate a particular thread to handle signals, or the OS may have a default policy on which thread should receive the signal.