Lecture Notes - Process, Threads and Scheduling

Debadatta Mishra
Indian Institute of Technology Kanpur

Process creation

- Enabling OS APIs to create new processes from the user space is an essential requirement of OS design. UNIX OSs provide fork( ) system call to create a new user process. The new process created (called the child process) inherits everything from the calling process (the Parent process). The fork( ) system call returns the child PID to the parent process and returns 0 to the child process.
- The OS performs a copy of the parent context---software state (PCB) and a copy of the hardware state. By implication, the memory pages are copied, page tables are set up afresh etc. In the software state, the PID is changed (a new PID is allocated) and a new OS stack is provisioned for the child process. The only change in the hardware context is to change the return value (stored in EAX register) of the newly created process. The child process is scheduled on a CPU and executes the next instruction (after fork syscall) on return from fork( ) system call. Note that, in this case, child is returning from a system call even if it never invoked one.
- When the child starts executing in user space
    - The stack is exactly same as the parent
    - Memory layout is exactly the same as parent
    - The code segments are exactly the same as parent
- The child process starts to deviate from the parent process only after this point. For example, if the child process dynamically allocates memory, they are mapped only to the child process. Similarly, if child executes a different set of function calls, the stacks of the parent process and the child process become different.
- Another system call required to allow creation of new user processes executing different binaries is exec( ). The exec( ) system call takes an executable as an argument, replaces itself (the caller) by the binary image provided in the argument. The exec system call cleans up the old process state (page table mappings, user pages etc.), creates a fresh virtual address layout with required physical memory allocations and loads the binary image from the hard disk. At the time of return to user space, the instruction pointer is set to the first instruction of the new executable. Apart from the PID and open file handles, all the traces of old executable are purged when the exec call returns. For more details and exact API specification, please refer to the man pages.
- The above mentioned system calls (fork and exec) can be used in tandem to create new user processes and self transit from a single process to many processes. For example, **init** process can create the shell as shown below,

```
main()
 {
    pid_t pid = fork( );
    if(pid  == 0){ /*child process*/
```

```
        exec("/bin/bash");
    }
    /*Parent process (init) logic*/
}
```

- Many times, the parent process may require to wait for the child to finish. For example, when a command is executed on the bash shell---using fork followed by an exec system call with the executable file corresponding to the command as an argument, the bash process should wait for the child process to terminate. wait( ) and waitpid( ) system calls provide the required waiting functionality. Refer to the man pages for the detailed specifications for wait( ) system call.
- Design of fork( ) inherently leads to a lot of process state replication which includes copying the parent process memory pages and the page tables. The question is: Can we avoid copying without hampering correctness.
    - Let us consider the requirement to copy the physical memory pages of parent and provide mappings in the child process. If both the parent process and the child process **READ** the memory pages, there should not be any requirement of creating a separate copy for the child process. Copy-on-write (CoW) is an optimization to implement fork( ) where the page table mapping of both child and parent processes point to the same physical page (originally of the parent) with a READ-ONLY access permission in the PTE. When the parent or the child process **WRITEs** to any memory location, a page fault is generated by the hardware, the OS creates a copy and updates the page table to allow writes.
    - Can we avoid copying the page tables to implement fork( ) if it is known that the child process invokes exec( ) shortly? If the parent process is suspended till the time child exits or invokes exec( ), then this scheme can work. Linux provides a fork variant named vfork( ) which implements the above semantic. Note that, any changes to memory allocation, content etc. performed by the child process before exit( ) or exec( ) will be visible to the parent process when it gets scheduled.
- Threads are software execution units of a process which can be scheduled on a CPU. Threads of the same process share almost everything apart from the CPU register state and the stack. As memory is shared across threads of a single process, minimal replication is required, and therefore, thread creation is faster than a process creation using fork( ).
- It is not necessary to have an one-to-one mapping of user's notion of a thread to the hardware context (schedulable entity in the OS, a.k.a. OS thread). A user space library can provide this abstraction to the programmer by mapping N threads to one OS thread or to M OS threads. In most current OSs, the mapping is one-to-one, which enables simple thread library implementations in the user space. POSIX threads (PThreads) is a well known thread library supported by most of the POSIX compliant OSs including Linux.
- Linux provides a single system call i.e., clone( ) to implement functionalities like fork( ), vfork( ) and pthread_create( ). The clone( ) system call provides the flexibility to the user to control the extent of sharing by specifying appropriate flags. For example,

pthread_create( ) invokes clone( ) with CLONE_VM flag to avoid memory and page table copy. Please refer to the man page of clone for more details.

Scheduling

- Unlike memory, a single CPU can not be divided into smaller units and shared across different processes. CPU is a time shared resource i.e., many processes own the CPU at different points of time. The CPU state can be saved into memory and restored by the OS; an essential characteristic required to implement time shared multiplexing. The question is, what are the states saved and restored on context switch when threads of a single process or threads of different processes are switched? Note that, even a normal process has one schedulable entity, normally referred to as the main thread.
- When switching between two threads of a same process, the page table mapping (pointed to by CR3 in X86 systems) need not be switched. By implication, TLB flush (even if ASID is not used) is not required in this case. However, the OS stack is required to be switched if an one-to-one mapping of user context to OS stack is to be maintained which is convenient to allow multiple threads of the same process to execute system calls concurrently. Most of the other OS states remain unchanged with the exception of thread ID.
- A process primarily joggles between the following three states during its lifetime.
    - Running state: The process is scheduled on CPU
    - Ready state: The process is waiting for OS scheduler to schedule it on CPU.
    - Waiting state: The process does not require CPU as it waits for I/O or other events.

  Most of the applications perform CPU operations for some time (a.k.a CPU bursts) followed by I/O operations in a cyclic manner. When a process in running state requests for I/O, it is moved to waiting state. After the event or I/O notification, the process is moved into ready state and becomes eligible for scheduling. Transition of a process from running state to ready state can occur if some other process is scheduled by the scheduler by switching out the current process.
- The OS scheduler has to select a process from the ready queue and load its context on a free CPU. However, it is possible that there may not be any process in the ready queue. To avoid this condition, most OSs have at least one process always in the ready queue called the “idle process”. In Linux kernel, this is commonly known as the swapper process. The idle process does not perform any task but it puts the CPU to a state where it can be woken up by external interrupts. In X86 systems, this can be achieved by executing HLT instruction which puts the CPU in a HALT state, woken up only by interrupts.
- Interrupts can be abrupt and deschedule the running process temporarily. One way to avoid interrupts taking over CPU is to disable interrupts, which is not recommended as disabling interrupts hampers the interactiveness of the system.
- Primarily, the scheduler can be invoked in the following conditions,
    1. A process enters the ready queue
    2. A process terminates
    3. A running process is required to wait for I/O or an event
    4. When the CPU receives an interrupt, specifically the timer interrupt

5. A process voluntarily invokes the scheduler (e.g., sched_yield( )).

Schedulers invoked only in second, third and fifth conditions are called non-preemptive schedulers. If the OS scheduler is also invoked in first and fourth conditions, it is called a preemptive scheduler.

- Non-preemptive schedulers are simple to design and implement. However, because of the reduced OS control, applications can dominate the CPU time causing issues like CPU starvation for other processes, unfair CPU sharing among processes, less interactive system etc.
- Preemptive schedulers provide greater flexibility to the OS to decide on CPU allocation. With a good scheduling strategy, the OS can overcome the above mentioned issues of non-preemptive scheduling. Preemptive schedulers are complex to design (as context switch has to be carefully carried out), incur higher scheduler overheads and may cause thrashing of CPU caches and TLB because of frequent context switching. Another aspect of preemptive scheduler design is to handle periods of no preemption of a process. For example, a process may not be switched out of the CPU if it is holding a critical resource which is also accessed by an interrupt handler.
- A scheduling policy should strive to optimize the following,
  - Maximize throughput (#of processes completed per unit time)
  - Minimize turnaround time (finish time - arrival time)
  - Minimize waiting time in the ready queue
  - Minimize response time (first schedule time - arrival time)

  Many a times, apart from the mean, standard deviation of the above metrics is desirable. For example, variance of response time and waiting time indicates the fairness of the scheduling policy.
- The scheduling problem can be simplified by treating each CPU burst as an independent process. For example, consider a process with arrival time at 2s, with a CPU burst between 2s-5s, followed by an I/O burst during 5s-15s and with a final CPU burst of 15s to 17s. This process can be treated as two scheduling requests arriving at time 2s and 15s for durations 3s and 2s, respectively.