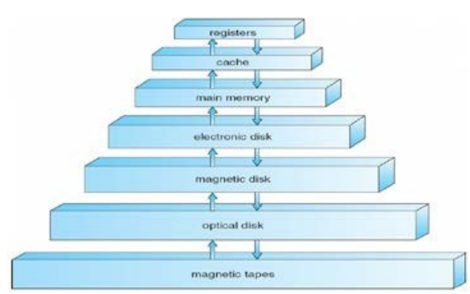
Study Guide to Accompany

Operating Systems Concepts *essentials Second Edition* by Silberschatz, Galvin and Gagne

By Andrew DeNicola, BU ECE Class of 2012

Introduction

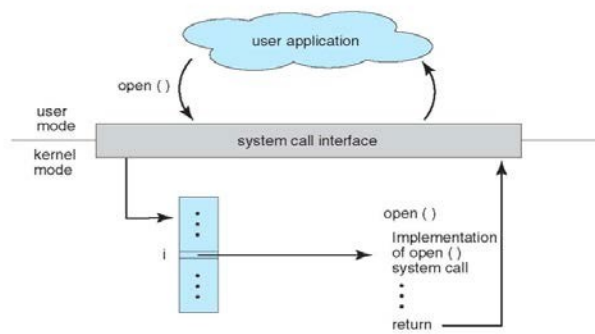
- An OS is a program that acts as an intermediary between a user of a computer and the computer hardware
- Goals: Execute user programs, make the comp. system easy to use, utilize hardware efficiently
- Computer system: Hardware ↔ OS ↔ Applications ↔ Users (↔ = 'uses')
- OS is:
  - Resource allocator: decides between conflicting requests for efficient and fair resource use
  - Control program: controls execution of programs to prevent errors and improper use of computer
- Kernel: the one program running at all times on the computer
- Bootstrap program: loaded at power-up or reboot
  - Stored in ROM or EPROM (known as firmware), Initializes all aspects of system, loads OS kernel and starts execution
- I/O and CPU can execute concurrently
- Device controllers inform CPU that it is finished w/ operation by causing an interrupt
  - Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines
  - Incoming interrupts are disabled while another interrupt is being processed
  - Trap is a software generated interrupt caused by error or user request
  - OS determines which type of interrupt has occurred by polling or the vectored interrupt system
- System call: request to the operating system to allow user to wait for I/O completion
- Device-status table: contains entry for each I/O device indicating its type, address, and state
  - OS indexes into the I/O device table to determine device status and to modify the table entry to include interrupt
- Storage structure:
  - Main memory – random access, volatile
  - Secondary storage – extension of main memory That provides large non-volatile storage
  - Disk – divided into tracks which are subdivided into sectors. Disk controller determines logical interaction between the device and the computer.
- Caching – copying information into faster storage system
- Multiprocessor Systems: Increased throughput, economy of scale, increased reliability
  - Can be asymmetric or symmetric
  - Clustered systems – Linked multiprocessor systems
- Multiprogramming – Provides efficiency via job scheduling
  - When OS has to wait (ex: for I/O), switches to another job
- Timesharing – CPU switches jobs so frequently that each user can interact with each job while it is running (interactive computing)



- Dual-mode operation allows OS to protect itself and other system components – User mode and kernel mode
  - Some instructions are only executable in kernel mode, these are privileged
- Single-threaded processes have one program counter, multi-threaded processes have one PC per thread
- Protection – mechanism for controlling access of processes or users to resources defined by the OS
- Security – defense of a system against attacks
- User IDs (UID), one per user, and Group IDs, determine which users and groups of users have which privileges

OS Structures

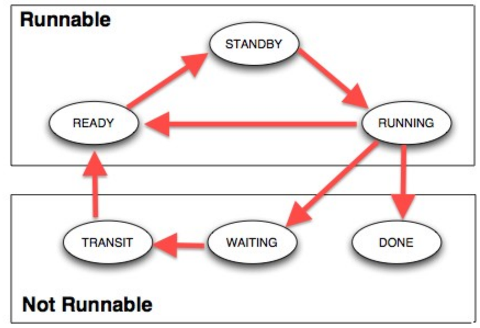
- User Interface (UI) – Can be Command-Line (CLI) or Graphics User Interface (GUI) or Batch
  - These allow for the user to interact with the system services via system calls (typically written in C/C++)
- Other system services that a helpful to the user include: program execution, I/O operations, file-system manipulation, communications, and error detection
- Services that exist to ensure efficient OS operation are: resource allocation, accounting, protection and security
- Most system calls are accessed by Application Program Interface (API) such as Win32, POSIX, Java
- Usually there is a number associated with each system call
  - System call interface maintains a table indexed according to these numbers
- Parameters may need to be passed to the OS during a system call, may be done by:
  - Passing in registers, address of parameter stored in a block, pushed onto the stack by the program and popped off by the OS
  - Block and stack methods do not limit the number or length of parameters being passed



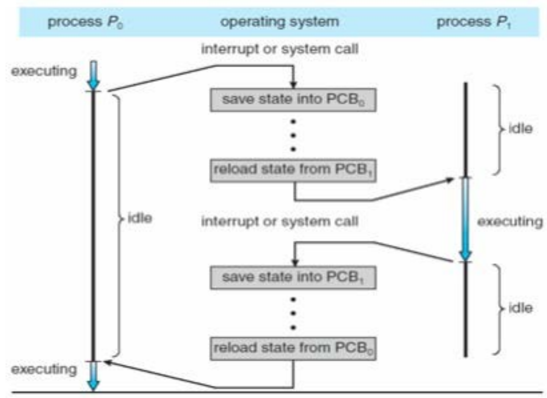
- Process control system calls include: end, abort, load, execute, create/terminate process, wait, allocate/free memory
- File management system calls include: create/delete file, open/close file, read, write, get/set attributes
- Device management system calls: request/release device, read, write, logically attach/detach devices
- Information maintenance system calls: get/set time, get/set system data, get/set process/file/device attributes
- Communications system calls: create/delete communication connection, send/receive, transfer status information
- OS Layered approach:
  - The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface
  - With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers
- Virtual machine: uses layered approach, treats hardware and the OS kernel as though they were all hardware.
  - Host creates the illusion that a process has its own processor and own virtual memory
  - Each guest provided with a 'virtual' copy of the underlying computer
- Application failures can generate core dump file capturing memory of the process
- Operating system failure can generate crash dump file containing kernel memory

Processes

- Process contains a program counter, stack, and data section.
  - Text section: program code itself
  - Stack: temporary data (function parameters, return addresses, local variables)
  - Data section: global variables
  - Heap: contains memory dynamically allocated during run-time
- Process Control Block (PCB): contains information associated with each process: process state, PC, CPU registers, scheduling information, accounting information, I/O status information



- Types of processes:
  - I/O Bound: spends more time doing I/O than computations, many short CPU bursts
  - CPU Bound: spends more time doing computations, few very long CPU bursts
- When CPU switches to another process, the system must save the state of the old process (to PCB) and load the saved state (from PCB) for the new process via a context switch
  - Time of a context switch is dependent on hardware



- Parent processes create children processes (form a tree)
  - PID allows for process management
  - Parents and children can share all/some/none resources
  - Parents can execute concurrently with children or wait until children terminate
  - fork() system call creates new process
    - exec() system call used after a fork to replace the processes' memory space with a new program
- Cooperating processes need interprocess communication (IPC): shared memory or message passing
- Message passing may be blocking or non-blocking
  - Blocking is considered synchronous
    - Blocking send has the sender block until the message is received
    - Blocking receive has the receiver block until a message is available
  - Non-blocking is considered asynchronous
    - Non-blocking send has the sender send the message and continue
    - Non-blocking receive has the receiver receive a valid message or null

Threads

- Threads are fundamental unit of CPU utilization that forms the basis of multi-threaded computer systems
- Process creation is heavy-weight while thread creation is light-weight
  - Can simplify code and increase efficiency
- Kernels are generally multi-threaded
- Multi-threading models include: Many-to-One, One-to-One, Many-to-Many
  - Many-to-One: Many user-level threads mapped to single kernel thread
  - One-to-One: Each user-level thread maps to kernel thread
  - Many-to-Many: Many user-level threads mapped to many kernel threads
- Thread library provides programmer with API for creating and managing threads
- Issues include: thread cancellation, signal handling (synchronous/asynchronous), handling thread-specific data, and scheduler activations.
  - Cancellation:
    - Asynchronous cancellation terminates the target thread immediately
    - Deferred cancellation allows the target thread to periodically check if it should be canceled
  - Signal handler processes signals generated by a particular event, delivered to a process, handled
  - Scheduler activations provide upcalls – a communication mechanism from the kernel to the thread library.
    - Allows application to maintain the correct number of kernel threads

Process Synchronization

- Race Condition: several processes access and manipulate the same data concurrently, outcome depends on which order each access takes place.
- Each process has critical section of code, where it is manipulating data
  - To solve critical section problem each process must ask permission to enter critical section in entry section, follow critical section with exit section and then execute the remainder section
  - Especially difficult to solve this problem in preemptive kernels
- Peterson's Solution: solution for two processes
  - Two processes share two variables: `int` **`turn`** and `Boolean` **`flag[2]`**
  - **`turn:`** whose turn it is to enter the critical section
  - **`flag:`** indication of whether or not a process is ready to enter critical section
    - `flag[i] = true` indicates that process P_i is ready
  - Algorithm for process P_i:

```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j)
                critical section
        flag[i] = FALSE;
        remainder section
} while (TRUE);
```

- Modern machines provide atomic hardware instructions: Atomic = non-interruptable
- Solution using Locks:

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```

- Solution using Test-And-Set: Shared boolean variable lock, initialized to FALSE

```
boolean TestAndSet (boolean *target){
        boolean rv = *target;
        *target = TRUE;"
        return rv:
}
```

```
do {
        while ( TestAndSet (&lock ))
                        ; // do
nothing
        // critical section
        lock = FALSE;
        // remainder section
} while (TRUE);
```

- Solution using Swap: Shared bool variable lock initialized to FALSE; Each process has local bool variable key

```
void Swap (boolean *a, boolean *b){
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

```
do {
        key = TRUE;
        while ( key == TRUE)
                        Swap (&lock,
&key );
        // critical section
        lock = FALSE;
        // remainder section
} while (TRUE);
```

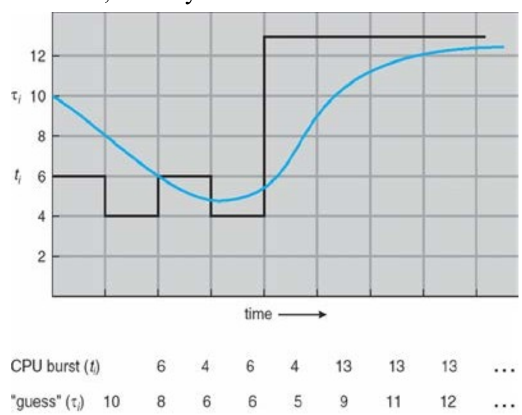
- Semaphore: Synchronization tool that does not require busy waiting
  - Standard operations: wait() and signal() ← these are the only operations that can access semaphore S
  - Can have counting (unrestricted range) and binary (0 or 1) semaphores
- Deadlock: Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes (most OSes do not prevent or deal with deadlocks)
  - Can cause starvation and priority inversion (lower priority process holds lock needed by higher-priority process)

Process Synchronization (Continued)

- Other synchronization problems include Bounded-Buffer Problem and Readers-Writers Problem
- Monitor is a high-level abstraction that provides a convenient and effective mechanism for process synchronization
  - Only one process may be active within the monitor at a time
  - Can utilize condition variables to suspend a resume processes (ex: condition x, y;)
    - x.wait() – a process that invokes the operation is suspended until x.signal()
    - x.signal() – resumes one of processes (if any) that invoked x.wait()
  - Can be implemented with semaphores

CPU Scheduling

- Process execution consists of a cycle of CPU execution and I/O wait
- CPU scheduling decisions take place when a process:
    - Switches from running to waiting (nonpreemptive)
    - Switches from running to ready (preemptive)
    - Switches from waiting to ready (preemptive)
    - Terminates (nonpreemptive)
- The dispatcher module gives control of the CPU to the process selected by the short-term scheduler
    - Dispatch latency- the time it takes for the dispatcher to stop one process and start another
- Scheduling algorithms are chosen based on optimization criteria (ex: throughput, turnaround time, etc.)
    - FCFS, SJF, Shortest-Remaining-Time-First (preemptive SJF), Round Robin, Priority
- Determining length of next CPU burst: Exponential Averaging:
    1. t_n = actual length of n^{th} CPU burst
    2. τ_{n+1} = predicted value for the next CPU burst
    3. α, $0 \leq \alpha \leq 1$ (commonly α set to 1/2)
    4. Define: $\tau_{n+1} = \alpha * t_n + (1-\alpha)\tau_n$



- Priority Scheduling can result in starvation, which can be solved by aging a process (as time progresses, increase the priority)
- In Round Robin, small time quantums can result in large amounts of context switches
    - Time quantum should be chosen so that 80% of processes have shorter burst times that the time quantum
- Multilevel Queues and Multilevel Feedback Queues have multiple process queues that have different priority levels
    - In the Feedback queue, priority is not fixed → Processes can be promoted and demoted to different queues
    - Feedback queues can have different scheduling algorithms at different levels
- Multiprocessor Scheduling is done in several different ways:
    - Asymmetric multiprocessing: only one processor accesses system data structures → no need to data share
    - Symmetric multiprocessing: each processor is self-scheduling (currently the most common method)
    - Processor affinity: a process running on one processor is more likely to continue to run on the same processor (so that the processor's memory still contains data specific to that specific process)
- Little's Formula can help determine average wait time per process in any scheduling algorithm:
    - $n = \lambda \times W$
    - n = avg queue length; W = avg waiting time in queue; λ = average arrival rate into queue
- Simulations are programmed models of a computer system with variable clocks
    - Used to gather statistics indicating algorithm performance
    - Running simulations is more accurate than queuing models (like Little's Law)
    - Although more accurate, high cost and high risk