INTRODUCTION

* OS = intermediary between user and hardware. Goals: 1. exe user programs; 2. Make comp. system easy to use; 3. Use hardware efficiently
* Computer System components: 1. Hardware > 2. OS > 3. Application programs > 4. Users
* OS is: 1. Resource Allocator (manages requests and fair resource use) ; 2. Control Program (prevents error and improper computer use)
* Kernel = the one program always running
* Bootstrap program = loaded at power-up
* System Call = request to the OS
* Device-status table = contains entry for each I/O device indicating its type, address, and state
* Interrupt =/= trap or exception (software generated interrupt caused by user request or error)
* Storage Structure: 1. Main memory (random access, volatile); 2. Secondary storage (non-volatile); 3. Disks (solid state - faster - or hard disks)
* Caching = copying information into faster storage system
* Device controller transfers blocks of data form buffer storage directly to main memory without CPU intervention
* Multiprocessor Systems are of two types: asymmetric or symmetric multiprocessing – each processor is assigned a specific task versus each processor performs all tasks.
* Dual-mode operation allows OS to protect itself and other system components – User mode and kernel mode
* Single-threaded processes have one program counter, multi-threaded processes have one PC per thread
* Timesharing = CPU switches jobs so frequently that each user can interact with each job while it is running
* Protection =/= security – controlling access of resources defined by the OS vs. defence against attacks
* Process = program in execution; unit of work within the system; active entity
* Process management activities: 1. Creating, suspending, resuming and deleting processes; 2. Provide mechanics for process sync and deadlock handling
* User IDs (UID) and Group IDs, determine which users and groups of users have which privileges
* Computing environments: 1.traditional 2.mobile 3.distributed 4.client-server 5.peer-to-peer 6.virtualization 7.cloud computing

OS STRUCTURES

* OS services: 1. User Interface (command-line, GUI, or batch); 2. Program Execution; 3. I/O operations; 4. File system manipulation; 5. Communication; 6. Error detection; 7. Resource allocation; 8. Accounting (keep track of how resources are used by each user); 9. Protection and security
* Most system calls (which usually have a number associated) are accessed by Application Program Interface (API) such as Win32, POSIX, Java
* Parameter passing to the OS: 1. Registers 2. Blocks 3. Stack (Block and stack methods do not limit the number or length of parameters being passed)
* SYSTEM CALLS: 1. Process Control (create/delete file, open/close file, read, write, get/set attributes) 2. Device management (request/release device, read, write, logically attach/detach devices) 3. Information maintenance (get/set time, get/set system data, get/set process/file/device attributes) 4. Information maintenance (get/set time, get/set system data, get/set process/file/device attributes)
* USER OS GOALS: convenient, reliable, safe and fast
* SYSTEM OS GOALS: ez to design, implement and maintain. Reliable, flexible and efficient
* Policy =/= Mechanism – decides what will be done vs how it will be done
* OS STRUCTURES: 1. Simple 2. Non-simple 3. Layered 4. Microkernel system
* 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 Components: 1. Text section (code) 2. Stack (temporary data aka local variables) 3. Data section (global variables) 4. Heap (memory dynamically allocated)
* A program becomes a process when .exe file is loaded into memory. One program can be several processes
* Process states: new, running, waiting, ready and terminated.
* Process Control Block (PCB): contains information associated with each process: process state, PC, CPU registers, scheduling information, accounting information, I/O status information
* Process types: 1. I/O Bound (more time doing I/O than computations, many short CPU bursts); 2. CPU Bound (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
* 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, returns 0 if child successfully created
* exec() system call used after a fork to replace the processes' memory space with a new program
* Whether the parent process or the child process executes first is undefined
* Interprocess communication (IPC) happens between cooperating (non-independent) processes.
* IPC options: 1. Shared memory 2. Message passing
* Producer-Consumer problem: fixed-size buffer, the Producer adds things to the buffer, the Consumer processes said things. Access to the buffer is mutually exclusive, producer cant produce if buffer full, consumer cant consume if buffer empty.
* Shared Memory issue: guarantee that processes’ memory accesses are synchronized
* Message passing requires a communication link. The implementation can be: 1. Physical (proc’s must call each other explicitly – send(P, mes) ) 2. Logical
* 2 types of message passing. 1. Blocking (synchronous – clingy, doesn’t move on without making sure the message was sent) 2. Non-blocking (asynchronous – they move on like normal)
* Ordinary vs named pipes – one-directional, must have parent child relationship vs bi-directional, no parent child relationship required.

PROCESS SYNCHRONIZATION

* Race Condition = several processes access and manipulate the same data concurrently, outcome depends on which order each access takes place. Concurrent data accesses can create data inconsistency
* Each process has critical section of code, where it is manipulating data
* Critical section problem = to design a solution that efficiently solves this problem. Each process must ask before entering its critical section
* Solution requirements: 1. Mutual exclusion (no 2 processes can be in their critical section at the same time) 2. Progress (make sure that we go down the list of processes waiting to enter the critical section) 3. Bounded waiting (bound imposed on the number of processes allowed to enter their critical section with others waiting)
* Modern machines provide atomic (non-interruptable) hardware instructions
* POSSIBLE SOLUTIONS:
  + Peterson’s solution: 2 variables – int turn (whose turn it is to enter CS) & Bool flag[n] (whether process is ready to enter CS or not)
  + Mutex Locks: first acquire(), then release() the lock
  + Semaphore: more sophisticated than mutex locks. Integer variable that can only be changed via wait() and signal()
  + 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)
  + 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