1 Introduction

- OS must efficiently allocate and share resources (data, programs, hardware)
- OS must support simultaneous parallel activity and switch between activities while ensuring safe concurrency
- OS events occur in an unpredictable order
- OS must offer persistent storage with easy to access files, access controls, failure protection and hardware management
- Kernel is always in memory and loaded on boot, implements commonly executed OS functions, and has complete access to hardware.
- Monolithic Kernels: Single black box with all functionality. Efficient calls within kernel and easier to write kernel components due to shared memory. But complex design with lots of interaction and no protection between components (driver can crash kernel).
- Microkernels: Little functionality in kernel as possible. IPC between servers (I/O, file, scheduling). Simple kernel less error prone. But overhead of IPC is high.
- Portable OS is one that can be ported from one system architecture to another with little modification.

2 Processes

- int fork() copies current process and returns twice: 0 in the child and child's PID in the parent
- int execve(const char *path, char *const argv[], char *const envp[]) runs new process located at path
- int waitpid(int pid, int* stat, int options) waits for pid=pid, -1(any child), 0(any child in the same process group) or -gid(any child in gid). Returns pid of terminated child or -1 on error normally.
- void exit(int status) terminates and returns exit status to parent process. int kill(int pid, int sig) sends sig to pid.
- int pipe(int fd[2]) returns fd[0] read end and fd[1] write end. Close one end before using with close(fd[i]).
- Processes can communicate via files, signals (kill process), pipes (blocks if full, SIGPIPE if closed), message queues, sockets, memory and semaphores.
- void signal(sig, my_handler) registers a signal handler for sig (cannot handle SIGKILL and SIGSTOP).
- Context switch: Direct cost of saving/restoring state + Indirect cost of pertubation of TLB/cache.

3 Threads

- Processes share address space, global variables, open files, child processes & signals. Threads have their own PC, registers & stack.
- int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routing) (void*), void *arg)
- void pthread_exit (void *value_ptr) makes value_ptr avaliable to any join with the terminating thread
- int pthread_yield(void)
- int pthread_join(pthread_t thread, void **value_ptr) blocks until thread terminates, value passed to pthread_exit in value_ptr's location.
- User-level threads: Kernel unaware of threads, managed by process. Process maintains thread table for thread scheduling. Better performance (thread creation/destruction/switching/synchronisation) but blocking syscalls stops all threads.
- Kernel-level threads: Threads managed by kernel. Blocking syscalls easily handled. But threads more expensive.

$$\text{CPU Utilisation } = 1 - p^n \text{ where p = prob that a tread is sleeping and n is the number of threads}$$

4 Scheduling

- Objectives: Fairness, Avoid indefinite postponement, Enforce priority, Maximise resource usage, Minimize overhead
- States: New, Ready, Running, Waiting, Terminated
- Metrics: Turnaround Time (Job submission to completion), Response Time (Request issued to first response)
- **First Come First Serve (Non-Preemptive)**: No indefinite postpoment. But lower avg turnaround
- **Round Robin Scheduling**: Run until blocks/quantum exceeds. Most fair. Avg turnaround low when run-times differ but poor for similar. Larger quantum causes smaller overhead but worse response time.
- **Shortest Job First**: Non-preemptive scheduling with run times known in advance. (Minimise average turnaround time)
- **Shortest Remaining Time**: Preemptive version of SJF where newly arrived processes can preempt running process.
- **Fair Share Scheduling**: Users assigned some fraction of CPU.
- **Priority Scheduling**: Can be externally defined by user or based on metrics. Can be static or dynamic.
- **General Purpose Scheduling**: Favour short and I/O bound jobs. Determine nature of job and adapt to changes.
- **Multilevel Feedback Queue**: One queue for each priority level. Starvation of lower-priority jobs, so increase job's priority as it waits. But does not react quickly to changes and cannot donate priority.
- **Lottery Scheduling**: Highly responsive, no starvation, jobs can donate tickets for common goal. But unpredictable.

5 Synchronisation

- No 2 processes may be simultaneously inside a critical section
- No process running outside a critical section may prevent another process entering a critical section
- No process requiring access to the critical section may be delayed forever
- No assumptions about relative speed of processes
- Strict Alternation: Busy wait and cannot execute loop twice in a row. Requires preemptive scheduling.

```
while (true) {
    while (turn != 0) {}
    critical_section(); turn=1;
    noncritical_section0();
}
```

```
while (true) {
    while (turn != 1) {}
    critical_section(); turn=0;
    noncritical_section1();
}
```

- Peterson's Solution:

```
int turn = 0; int interested[2] = {0,0};
void enter_critical(int thread) {
    int other = 1 - thread;
    interested[thread] = 1;
    turn = other;
    while (turn==other&&interested[other]) {}
}
```

```
void leave_critical(int thread) {
    interested[thread] = 0;
}
enter_critical(0);
critical_section();
leave_critical(0);
```

- Busy wait: Efficient if short wait time, thread runs immediately after wait finished, not wasteful if less threads than processors. But wastes CPU if wait is long.
- Spin Locks: Busy wait, Priority Inversion problem

```
void lock(int L) {
    while (TSL(L)!=0) {}
}
```

```
void unlock(int L) {
    L = 0;
}
```

- Race Condition occurs when multiple threads r/w shared data and final result depends on relative timing of execution
- Semaphore stores a counter and a list of processes in the queue. Can be used for mutex or ordering events.
- Producer-Consumer model with 3 semaphores: Mutex, Not Full, Not Empty.
- Monitor is a language construct. Only 1 process can be in a monitor at a time. wait(c) waits for c to be signalled, signal(c) wakes 1 process and broadcast(c) wakes up all processes. If signal with no one waiting, it is lost.
- Use waitpid(pid, NULL, 0) or pthread_join(thread, NULL) to force one process/thread to execute first.

6 Deadlock

- **Conditions**: (1) Mutual Exclusion: Each resource is either available or assigned to exactly 1 process
- (2) Hold and Wait: Process can request resources while it holds other resources earlier
- (3) No Preemption: Resource given to a process cannot be forcible revoked
- (4) Circular Wait: Processes can wait for a resource held by the next process in a circular chain
- **Solutions**: Ignore it. Good for infrequent deadlocks (low contention for resources).
- Detection & Recovery: Dynamically build resource allocation graph and check for cycles. Recovery include pre-emption (temporarily take resource from owner and give another), rollback (previous checkpoint state) or killing random processes.
- Dynamic Avoidance: System only grants resources when it knows it is safe
- Prevention: Eliminate one of the four conditions. (2) Request resources atomically before start (global lock), but must know all in advance. (3) Threads give up locks voluntarily if not all locks can be obtained. (4) Order lock acquisition based on memory address of lock.
- **Livelock** is when processes are not blocked but system as a whole not making progress.
- **Communication Deadlock** when message is dropped, so use timeouts instead
- **Detection:** A holds X and wants Y. Point from X to A and A to Y.

7 Memory Management

- Objectives: Memory Allocation & Memory Protection
- Requirements: No knowledge of how address are generated & No knowledge of what addresses are used for
- Memory Management Unit is a hardware device in CPU for mapping logical to physical addresses managed by kernel (define mapping, manage page table, page faults) and hardware (TLB).
- Kernel: low memory (3-3.896GB logical space, 3.896-4GB for on-demand mapping). User: high memory (0-3GB logical space).
- Relocation registers: Base register contains smallest physical address. Limit register contains range of logical address.
- Multiple-Partition Allocation: OS maintains info about allocation partitions and holes. When new process arrives (1) First-fit (2) Best-fit, smallest hole high enough (3) Worst-fit, largest hole.
- External fragmentation: total memory exists to satisfy request but not contiguous, reduce by compaction. Internal fragmentation: Allocated memory larger than requested.

- Paging: To run program of size n pages, find n free frames, load program and set up page table. No external fragmentation.
- Address Translation: For logical space of 2^m and page size 2^n, page number (index into page table) m-n bits and page offset (to combine with base address from page table) n bits
- Page Table: Page Table base register points to page table and Page Table Length Register indicates size
- Bits for each page table entry: size of physical memory / page size (frame number) + protection (demand paging) & reference (clock algo) bits. Need not store page number as it is the index of the page table.
- Max entries in page table: Logical address space / page size
- Memory Protection: Attach valid-invalid bit to each page table entry (page fault - demand paging/segmentation fault).

For n-level page table, $EAT = (\epsilon+t)\alpha+(\epsilon+(1+n)t)(1-\alpha)$, where t = memory cycle time, ϵ = associative lookup time, α = hit ratio

- Hierarchical Page Table, Hashed Page Table (page table contains chain of elements with same hash) or Inverted Page Table (entry in page table for each memory frame: value = pid + page number, index = frame number, all processes share 1 page table).
- Shared Memory: No need for kernel involvement after establishment. Useful for IPC and sharing libraries.
- Segmentation: Seperate address space for code, data, stack. But memory allocation harder due to variable size (ext frag).

$$EAT = (1-p)(\text{memory access}) + p(\text{page fault/restart overhead} + [\text{swap page out}]/\text{in})$$

- Copy-on-Write: Parent and child share pages unil either process modifies a page.
- Memory-mapped files: Map file into virtual address space using paging.
- First-In-First-Out: But may replace heavily used page (Belady's Anomaly - more frames but more page faults).
- Optimal Algorihtm: Replace page that won't be used for longest period of time. Impossible in practice but benchmark.
- Least Recently Used: Each page has a counter, replace lowest counter. But expensive so use approximate.
- Second Chance (Clock) Algorithm: Each page has a reference bit (initialised to 0), pointer to next victim.
- Counting Algorithm: Keep counter of reference to each page. Replace page with smallest(age old counters)/largest count.
- Thrashing: Excessive paging activity causing low processor utilisation, instead processor repeatedly pages from disk.
- Working Set Clock Algorithm: Second Chance (Clock) Algorithm but only evict page if age > working set age. But how to determine size of working set, if too many page faults allocate more page frames.
- Global strategy (Linux): Memory shared between all processes. Initial proportional to process size, page fault frequency to tune allocation. Local strategy: each process gets fixed allocation of memory.

8 Device Management

- Objectives: Fair access to shared devices (Allocation of dedicated devices), Exploit parallelism of I/O devices for multiprogramming, Provide uniform simple view (Hide complexity of device handling, Uniform naming and error handling)
- Character: Data in stream of bytes. Block: Data in fixed size blocks (Cache - can be bypassed with Direct I/O for kernel/databases).
- I/O Layers: User-level I/O software, Device-independent OS software, Device drivers, Interrupt handlers, Hardware
- Device allocation: Dedicated vs Shared. Spooler daemon to share nonsharable devices.
- Buffered I/O used to smooth peaks in I/O traffic. Unbuffered I/O transferred directly, high process switching overhead.
- Programmed I/O: CPU constantly checks if I/O is ready
- Interrupt-Driver I/O: I/O module will interrupt CPU when ready.
- I/O using Direct Memory Access: DMA engine transfers data between devices and main memory without CPU.
- Linux I/O classes: Character (unstructured), Block (structured), Pipes (unidirectional), Socket (bidirectional)
- fd = create(filename, permission). fd = open(filename, mode - 0 read, 1 write, 2 read/write). close(fd). numbytesread = read(fd, buffer, numbytes). numbyteswritten = write(fd, buffer, numbytes).
- Each process has own fd table with initial: 0 stdin, 1 stdout, 2 stderr, referring to terminal where program started
- (Operations) Blocking I/O: process blocked until I/O completed. Non-blocking I/O: I/O returns as much as possible
- (Requests) Asynchronous I/O: Process execute in parallel with I/O operation, process asks if any is ready/callback.

9 Disk Management

- Cylinder is made up of tracks (rings), made up of sectors.
- Access Time = Seek time (move head) + Latency Time (rotate disk) + Transfer time =

$$t_{seek} + \frac{1}{2r} + \frac{b}{rN}, \text{ where b = bytes transferred, N = bytes/track, r = rotations/second}$$

- First Come First Serve: Random seek patterns. Ok for lightly loaded disks only.
- SCAN Scheduling: Choose request that results in shortest seek time in current direction, change only at ends.
- Linux: SCAN algorithm + Deadline scheduler + Anticipatory scheduler (delay after read request for spatial locality)

- RAID Level 0 (Striping): Spread data across multiple disks. Concurrency but no redundancy.
- RAID Level 1 (Mirroring): Mirror data across disk. Faster read, slower write, easy failure recovery, high overhead.
- RAID Level 5 (Block-Level Distributed XOR): Distributed parity info. Reconstruction of failed disk slow.
- Disk Cache (In Memory): LRU / LFU (blocks may be referenced many times in short period of time - misleading count)
- Frequency-Based Replacement: New and Old stack. Referenced block to top of stack. Only increment reference in Old.

10 File Systems

- Objectives: Storage, Sharing, Concurrent access, Organisation & management
- File space allocated in blocks (512-8192B). Large block - int frag, small blocks - high overhead for large files.
- Contiguous File Allocation: But external fragmentation and poor performance if file grows and resize needed
- Block Linkage (Chaining): But blocks disperse throughout disk, search process slow (seek time).
- Block Allocation Table: Directory entries indicate first block in BAT. BAT stores index of next block. Less seek time.
- Index Block: Each file has one or more index blocks containing list of pointers to data blocks + chaining
- (Linux) Inodes: Type & access control, no. of links, user ID, group ID, access time, modify time, inode change time.
- Free List: Linked list of blocks containing locations of free blocks. But files allocated in noncontiguous blocks.
- Bitmap: Contains bit for each disk block to indicate usage. Can determine contiguous blocks but may need searching. Located in first block of storage.
- (Linux) Directory Representation: Directory entry (inode no + directory entry len + file name len + file name)
- Links: Hard Link - reference address of file (not dir), Soft Link - reference path of file/dir
- Mounting: Mount table (location of mount points & devices). Combine multiple FS into 1 namespace. Support soft-links to files in mounted FS but not hard links (since cannot reference inode of another FS).
- (Linux) File: open, close, read, write, lseek, stat, fcntl. Dir: mkdir, rmdir, link, unlink, chdir, opendir, closedir, readdir, rewinddir
- (Linux) ext2fs: Block groups of contiguous blocks. Contains superblock (info about FS), group descriptor (info about block group), block allocation bitmap (which blocks in block group in use), inode allocation bitmap (which inode in block group in use), inode table (array of inodes), data blocks (actual data). Inode (128B) has 12 direct, 1 indirect, 1 double-indirect and 1 triple indirect.

11 Security

- Access Control Matrix: Access Control List (column, per object) vs Capabilities List (row)
- Set User ID (SUID) bit: Switch effective UID to file owner when executed. (Real/Effective/Saved UID)
- Bell-La Padula Model: Read down, write up (confidentiality). Biba Model: Read Up, write down (integrity).