CSCC69 – Operating Systems
University of Toronto Scarborough, Winter 2019

Operating Systems and Fundamentals

Operating System – the software layer between user applications and hardware
Resource Manager – allows proper usage and allocation of resources
Control program – controls execution of user programs to prevent errors and improper use
Does: Synchronization, Scheduling, Memory management, File systems, networking, machine code
Storage Hierarchy – CPU > L2/L3 Cache > Memory > Disk > Tape (faste r to cheaper)
Operating systems are **event driven programs** - there are constant interruptions and requests and context switches

Process – a job or a unit of work. It can be a program in execution. It contains all state of program in execution
- An **address spaces**
- Referred to by its process ID (**pid)**
- Contains all of its OS resources and general purpose registers
- Represented by **Process Control Block (PCB)**
    - All the info about a process
    - Dynamically allocated, when process is created OS allocates a PCB for it and places on ready queue
    - This is all the info needed to restore the hardware to same config as process was
        - Process State (ready, running blocked)
        - Stack pointer (last program request)
        - Program counter (register containing address of next instruction)
        - Registers
        - CPU scheduling info
        - Memory management (page tables)
        - I/O status
- Can be created by: batch jobs, user requests, by other processes, system initialization
- Can be terminated by: last statement of code, parent terminates child, errors

Queues – OS maintains a collection of queues that represent the state of all processes (ready, waiting, etc.), processes are moved from queue to queue by PCB as the process executes
Context Switch – switching the CPU from one process to another, saving state for old process and loading state for new (bottleneck, so we're using threads to try to avoid this)

int fork() – creates a new address space with a copy of entire contents of address space of parent – returns twice (child PID to parent, 0 to child)
Inefficient – making multiple copies of the same data, code, etc. We should use **threads**! Have multiple SPs, PCs, that define threads pointing to different parts of the same process. Thus **processes** are containers for thread exc
Pthread API – pthread_create, join, cancel, exit define thread operations for the PThread api.

Process Dependence –
Independent – if it cannot affect or be affected by other processes executing in the system
No data sharing -> process is independent
Cooperating – if the process is not independent. They must be able to communicate with each other and synchronize their actions
Can communicate using shared memory or message parsing (send, receive model)

System Calls and Synchronization

BIOS – Basic Input Output System, small program stored in hardware's non-volatile memory
Bootstrapping – The BIOS runs a routine when the computer starts
- Check attached peripherals (keyboard, RAM, etc.)
- Checks connected and new devices
- Determines boot order
- Primary bootloader reads partition table and loads secondary bootloader
- Secondary bootloader loads OS into memory

Machine Starts In System Mode so that kernel code can execute immediately. **Initialize Data Structures -> Create first Process -> Switch to User Mode and run process -> Wait for something to happen**
int exec(char *prog, char *argv[]) – stops current process, loads prog into process' address space, initializes hardware context and args, places PCB onto ready queue – DOES NOT CREATE NEW PROCESS

USER > Kernel: Boot time, explicit system call, hardware interrupt, software traps, etc.
Major system calls: **Process:** fork(), waitpid(), execve(), exit(status), **FS:** open, close, read, write, lseek, stat, etc.

Interleaved Schedules – some execution of processes can be interleaved, resulting in undesired results. This is from shared resources (variables, counters, registers, etc.)
Race condition – outcome depends on the order in which accesses take place
Synchronization - ensure that only one thread at a time can manipulate the shared resource(s)
Critical Section (CS) Problem – designing a protocol that threads can use to cooperate using shared resources

1) **Mutual Exclusion** If one thread is in the CS then no other is
2) **Progress** If no thread is in the CS and threads want to enter CS, it should be able to enter in definite time
3) **Bounded Waiting** If some thread is waiting on CS, limit number of times other threads can enter before it can
4) **Performance** The overhead of entering and exiting the CS is small wrt the work being done in it **(minor)**

Turn Solution – progress not satisfied, excluded from entering even if other process is out of CS
Flags Solution – mutual exclusion not satisfied, may both read {false, false} and both enter CS
Peterson's Solution/Algorithm – solution for achieving mutual exclusion using **Turn** and **Flags** (turn, interested[N])
Hardware might need changes – **disable interrupts and context switches**
Test-and-Set Lock – test_and_set(Boolean *lock) acquires lock, switches 0 > 1. Basically returns old value but updates variable to some non-zero value. **Always True** - Either it was True (locked) already, and nothing changed or it was False (available), but the caller now holds it

Locks

Very primitive, minimal semantics. **Only allows one thread to enter** CS.
Spinlock – acquire() and release(), uses **busy waiting,** while loop in acquire() – consumes CPU cycles

Semaphores

Basic, easy to understand, hard to program with. **Allows X numbers of threads to enter** CS. System wide – shared by multiple processes. **Mutex** are Binary ones – **Counting** can accept > 1 process at a time.
Producer – Consumer Problem – (producers make, consumers take, but what if make while full or take while empty?) Solution is **Sleep/Wake**.
Semaphores have **wait** (decrement avail variable) and **signal** (increment avail variable) **– but this is CS problem!**
Semaphore **w_or_r** - exclusive writing or reading
int **readcount** - number of threads reading object - to detect when a reader is the first or last of a group.
Semaphore mutex - control access to readcount

Monitors

High-level, ideally has language support (Java)
Protects regions – not just variables or resources.
Enters monitor by invoking one of its procedures - Other processes are blocked from entering it
May need to wait – allowing other processes to use monitor. Works using a variable called a **condition**
Wait() and Signal()
Monitor Semantics:
- **Hoare Monitors –** signal switches from caller to waiting thread, need queue for signaler if not done using mntr.
- **Brinch Hansen** – signaler must exit monitor immediately, signal() is always last statement in monitor procedure
- **Mesa Monitors –** signal places a waiter on the ready queue, but signaler continues inside monitor

Messages

Simple model for communication & synchronization
Direct application to distributed systems

Dining Philosophers Problem – needs two forks to eat, getLeft and getRight but it will block each other indefinitely.
Solved using **mutexes →**

```
void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = THINKING;                /* philosopher has finished eating */
    test(LEFT);                         /* see if left neighbor can now eat */
    test(RIGHT);                        /* see if right neighbor can now eat */
    up(&mutex);                         /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Process Scheduling

Only one process can run on CPU. We want to give illusion that processes are running concurrently. Maximize CPU usage.
Process Scheduling – The allocation of processors to processes over time
Multiprogramming – increase CPU utilization and job throughput by overlapping I/O and computation
Scheduling Goals –

- Fairness – each process receives **fair share of CPU**

- Avoid Starvation of processes
- Poly enforcement – usage policies must be met
- Balance – all parts of the system should be busy

Batch Systems

- Throughput – maximize jobs completed per hour
- Turnaround time – minimize time between submission and completion
- CPU utilization – always busy CPU

Non-preemptive scheduling – once CPU has been allocated to process, keeps CPU until it terminates. For batch scheduling

FCFS (First Come First Served): Choose the process at the head of the FIFO queue of ready processes

- Average waiting time usually quite long
- **Convoy effect:** all other processes wait for the one big process to release the CPU

Shortest Job First – choose the process with the shortest processing time

- Need to know processing time in advance
- Unfair, potential starvation

SJF has unfairness, FCFS has long wait times – what if we allow preemption?

- Don't always run jobs to completion
- Preempt a job that's running too long

Preemptive scheduling – CPU can be taken from a running process and allocated to another. Real-time or interactive

Round-robin: ready queue is circular, each process runs for time quantum Q before preempted back onto queue
As q->inf RR -> FCFS, as q->0 RR -> processor sharing (PS). Want q large wrt. Context switch time

Priority Scheduling – priority p is given to each process, highest priority job is selected from Ready queue

- Can have starvation (low p job)
- A low priority task may prevent higher priority task from making progress **priority inversion**

Multi level queue scheduling

- First policy determines which queue(s) to serve (priority)
- Second decides which job within a queue to choose (FCFS, RR)
- Highest priority gets 1q, second gets 2q – after 1q is done, bumped to 2nd priority – this way shortest highest priority finishes first

Feedback Scheduling

- Want to give priority to shorter jobs, IO bound jobs, interactive jobs, etc. but don't know type of job before
- Combine with MLQ to prefer jobs that finish faster, change priority of processes based on age, CPU usage, etc
- If there are tasks with same priority, use RR to rotate them. Switch after **granularity** time units.

Assigning priority – done through nice() call – assigns the priority of processes from [-20, 19] with negative numbers being more important.

Interactive/IO should be favored – give bonus on top of static priority. Identified by time process sleeping vs CPU time

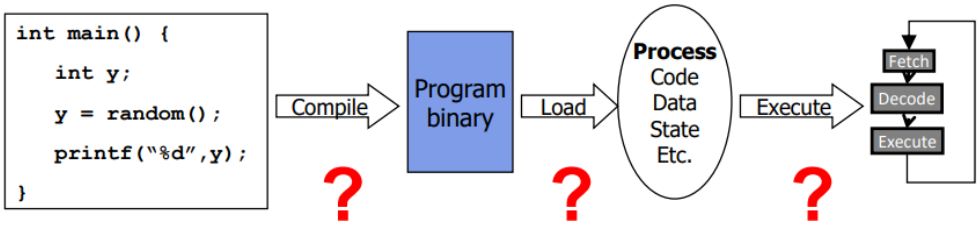
Mars rover issue – low priority (gather data) is blocked by higher priority (bus management) and can't call med priority (data sending) so no tasks are accomplished.

Real Systems – combination of MLQS and Feedback scheduling

Memory Management

Address Translation – translate a virtual address to a physical address on disk/memory/etc.

Address Binding – program binary is loaded into process, translated from variable names into real memory addresses



Addresses are bound: Compile time (absolute code, needs to know what memory needs to be used and no relocation) vs. Load time (resolves references in other files, but also can't be moved). Best plan is **at Execution time.**

Translation – symbolic addresses (var names) →logical addresses (relative to start of stack) → physical addresses

Allocation of Phys memory:

- Efficient management
- Don't let memory go wasted
- Entire process must be in memory to run

External Fragmentation – Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous so it can not be used.

Internal Fragmentation - Memory block assigned to process is bigger. Some portion of memory is left unused as it can not be used by another process.

Fixed Partitioning – divide memory into regions with fixed boundaries (eq or uneq sizes are ok). **Disadvantage:** Internal fragmentation, must deal with programs that are larger than partitions (**overlays)**

Placement w/Fixed Partitions – have queues that wait on partitions, if equal sized processes can wait at any partition, if unequal, then assigned to smallest partition in which it will fit

Dynamic Partitions – a partition of exactly the right size is created to hold it – holes are created (**external fragmentation)**

Malloc() and Free() – allocates contiguous range of logical addresses, free just releases the allocated block of ptr

Tracking Memory Allocation

- **Bitmaps** (1 == allocated, 0 == free), requires scanning bitmap for sequence of N zeroes, and is slow
- **Free lists** (maintain LL of allocated/free) **Implicit List** (each LL node has header of size and status (a/free))
  Explicit list (DLL list of pointers in free blocks)

Adjacent free blocks can be coalesced (5free + 4free = 9free), would be easier if they had a **boundary tag**

Placement Algorithms:

- **First-fit** - choose first block that is large enough; search can start at beginning, or where previous search ended (called next-fit) [Simple, fastest, most efficient, small fragments near start though]
- **Best-fit** - choose the block that is closest in size to the request [left-overs are small and unusable, same FF]
- **Worst-fit** – choose the largest block [bad.]
- **Quick-fit** – keep multiple free lists for common block sizes [great for fast allocation, hard to coalesce]

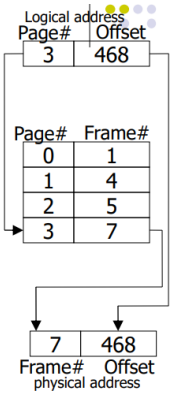
To translate virtual address: 0x3468

- Extract page number (high-order 4 bits)
  -> page = vaddr >> 12 == 3
- Get frame number from page table
- Combine frame number with page offset
  - offset = vaddr % 4096
  - paddr = frame * 4096 + offset
    - paddr = (frame << 12) | offset

Bit 15 12 11 0

0011 | 0100 0110 1000

Pg # Offset



Paging as a Solution – partition memory into equal fixed-sized chunks (**Page Frames, Frames)**

Divide process' memory into chunks of same size (**Pages)**

Eliminates External Fragmentation, reduces **Internal Fragmentation** to one page per process

Relocations – Done by the MMU (Memory Management Unit)

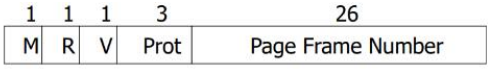
Base Register – base register is loaded with starting address of process

Limit Register – the last address of process

Page Tables – The virtual addresses are now page_number + page_offset, as below:

Page Number = vaddr/page_size Page Offset = vaddr % page_size

Simpler to calculate if page_size is pow of 2



- Page table entries (PTEs) control mapping
  - Modify bit (M) says whether or not page has been written
    - Set when a write to a page occurs
  - Reference bit (R) says whether page has been accessed
    - Set when a read or write to the page occurs
  - Valid bit (V) says whether PTE can be used
    - Checked on each use of virtual address
  - Protection bits specify what operations are allowed on page
    - Read/write/execute
  - Page frame number (PFN) determines physical page
  - Not all bits are provided by all architectures

Suppose addresses are 16 bits, pages are 4K (4096 bytes)
How many bits of the address do we need for offset? **12 bits** (2^12 = 4096)
What is the maximum number of pages for a process? **2^4**

Page Lookups

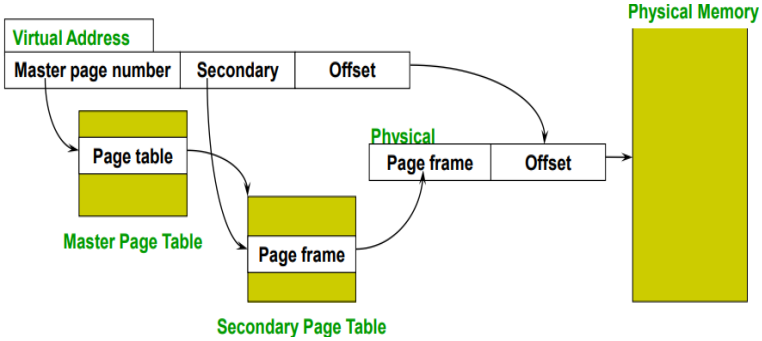
Normal -> Page Number refers to Page Table, offset is found in Physical Memory

- This approach needs 2 ref (page table, phys memory)
- Idea: Use TLB (Translation lookaside buffer (virtual pg# to PT entry))
- Small hardware cache of recently used translations

Cache tags are virtual page numbers, cache values are page table entries (entries from page tables, not phys address)
Can directly calculate physical address with PTE + offset

Only 64 – 1024 entries, but still handles >99% of translations (but there are still misses). Exploits locality. Managed by:

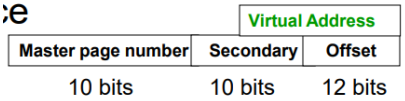
- MMU (Hardware), knows where PTs are in main memory, MMU accesses them directly,
- OS (software) finds appropriate PTEs and loads into TLB, keep consistency, evict entries, etc.



This gets large. We can try to only map the fraction of address space being used. We can use **two-level page tables.**

Master page →Page in SPT→ Page Frame → Offset → Physical memory address

Inverted PTs – one table with entry for each physical page frame, entries record which virtual page# is stored in that frame. Less space, but lookups are slower



Page Allocations and Evictions

When it evicts a page, the OS sets the **PTE as invalid** and stores the **location of the page in the swap file** in the PTE
If a process accesses an evicted page → **Load Page** in memory, **Update PTE**

- Causes a trap (**Page Fault)**
- Trap will run OS page fault handler
- Handler uses invalid PTE to locate page in swap file
- Reads page into a physical frame, updates PTE to point to it
- Restarts process

Fetch Policies – when to get a page **Demand paging** (when we need it) vs **Prepaging** (Predict future page at current fault)
Placement Policies – where to put a page **NUMA** (nonuniform memory access) and **Cache** performance are factors
Replacement Policy – what page to evict **done based on a REFERENCE STRING,** causing the least amount of page faults

Cold Misses – first access to a page (unavoidable)
Capacity Misses – caused by replacement due to limited size of memory

Belady's algorithm is known as the optimal page replacement algorithm because it has the lowest fault rate for any page reference stream (aka OPT or MIN). **It is not doable because you need to know the future perfectly. It is a metric.**
Algorithms: (+ means easy to implement) (* means good performance)

+NRU (not recently used) – Divide into 4 classes, remove from lowest numbered class that's not empty

- Class1: Not referenced, not modified
- Class 2: Not referenced, modified
- Class 3: Referenced, not modified
- Class 4: Referenced, modified

+FIFO (first in first out) - Maintain a list of pages in order in which they were paged in, evict oldest page

- Suffers from **Belady's Anomaly** →The fault rate might actually increase when the algorithm is given more memory (very bad)

+*Second Chance/Clock – evict the oldest page that has not been used in a circular clock by using reference bit
If the ref bit is on turn it off and advance
If the ref bit is off evict page and advance

****LRU** (least recently used) - On replacement, evict page that has not been used for the longest time in the past

- Difficult to implement, either needs to double PTE size for info or use stack, lots of operations
- We implement **approximate LRU** instead. All R bits are 0, then set to 1 for use. Shift R bit into high bit, shift other bits to the right. Pages with larger counters are more recently used (do this periodically).

LFU/MFU (least/most frequently used). Neither is common, both are poor approx.. of OPT

A working set of a process is used to model the dynamic locality of its memory usage defined by Peter Denning
Intuitively, want the working set to be the set of pages a process needs in memory to prevent heavy faulting
Hard to answer so its not a Page Replacement Algorithm, but its good abstraction
Page Fault Frequency (PFF) is a variable space algorithm that uses a more ad-hoc approach – more faults give it more memory, less faults take away memory. However subjet to **Belady's Anomaly**
Thrashing - When more time is spent by the OS in paging data back and forth from disk than executing user programs

File Systems

File Systems - They provide a nice abstraction of storage. Implements abstraction (files) for secondary storage, and logical organization (directories), permits sharing of data between processes, people, and machines and security
Files have the following attributes:

- Name, Owner, Location, Size, Protection, Creation time, Time of last access

Sequential access – reads bytes one at a time, read/write next
Direct access – access given block/byte number read/write n
Open-file table – system-wide record of open files
There are actually 2 levels of internal tables

- a per-process table of all files that each process has open (this holds the current file position for the process)
- each entry in the per-process table points to an entry in the open-file table (for process independent info)

Directories – list of entries, names and associated metadata. Can **Search, Create, Delete, List,** and **Update**
Path Name Translation - /usr/bin/c → get root, get loc of usr, open user, get loc of bin, open bin, get loc of c, open c
Caching – OS will cache prefix lookups for performance /a/b, /a/bb /a/bbb, etc.
Symbolic/Soft Link – directory entry that contains true path to a file (deletion of links is ok, deletion of file = dangle ptrs)
Hard Link – second directory entry identical to the first (keep a reference count)
File Protection – None, Knowledge, Execution, Reading, Appending, Updating, Change Protection, Deletion (R/W/X UNIX)
Access Control Lists for each object, maintain a list of subjects and their permitted actions (easier to manage)
Capabilities for each subject, maintain a list of objects and their permitted actions (easier to transfer)

Storing Files and Implementing a File System
File Systems define a **block size** (e.g. 4KB)
Master Block determines root directory (superblock, partition control block) – always at well known location, replicated
Freemap determines which blocks are free, stored as a bitmap, one bit per block, cached in memory stored on disk
Remaining blocks are used to store files and directories

Directory Implementations
- **Linear List** – simple list of file names and pointers to data blocks, needs linear search and slow to execute
- **Hash Table –** hash file name to get pointer to entry in linear list (above)

File Storage
- **Contiguous Allocation –** fast, easy, but inflexible and causes fragmentation
- **Linked/chained structure –** good for sequential access, bad for all others
- **Indexed structure** – index block contains pointers to many other blocks, may need multiple blocks linked together

To open "/one", use Master Block to find inode for "/" or disk and read inode into memory
inode allows us to find data block for directory "/"
Read "/", look for entry for "one"
This entry gives locates the inode for "one"
Read the inode for "one" into memory
The inode says where first data block is on disk
Read that block into memory to access the data in the file

Indexed Allocation – UNIX Inodes
Unix inodes implement an indexed structure for files
- Each inode contains 15 block pointers
- First 12 are direct block pointers (e.g., 4 KB data blocks)
- Then single, double, and triple indirect

They are **not** directories, only describe where on the disk the blocks for a file are placed (directories are files)

File Buffer Cache – cache file blocks in memory to capture locality (common), even small sizes can be effective
Writes are slow → use Log structured file system, always write contiguously at end of previous write
Read Ahead → FS predicts that the process will request next block. Big win for sequentially accessed files
- FS goes ahead and requests it from the disk, this can happen while the process is computing on previous block

Disks and I/O

Sequential access is a lot faster than **random access** due to how the disk seeks and spins
Disk Request Performance
- Seek (move arm to correct cylinder)
- Rotate (wait for sector to rotate under head)
- Transfer (transfer data from surface to disk controller to host)

Disk I/O is slow so minimize disk access (cache) and try to minimize disk access cost (request scheduling, sequential)

Need to know a lot for disk requests – cylinder, surface, track, sector, size etc. We use **logical block addressing**
- **Disk exports data** as logical blocks 0…N
- **Only need to specify logical block #**

Disk Scheduling
- FCFS – First come first serve (reasonable when low load, long wait times for long queues)
- SSTF – Shortest seek time first (minimize arm movement, favors middle blocks)
- SCAN – also elevator, service requests in one direction then reverses
- C-SCAN – only serves requests in one direction
- LOOK/C-LOOK – only goes as far as the last request in each direction (not whole disk)

Modern Disks do disk scheduling themselves, knows itself better than OS does.
Problems with Original UNIX FS – fixed by **Fast FS** by using **Cylinder Groups**, original problems were that
- Data blocks allocated randomly in aging file systems
- Inodes are far from blocks

Linux Second Extended File System (EXT2)
- ext2 uses block groups (like cylinder groups, but with more information like group descriptor)
- directories are files too
- order should go, write the data first, then write the inode, then add to directory

Log Structured File System (LSF)
- Write all FS data in a continuous log
- Uses inodes and directories from FFS
- Needs inode map to find inodes

New Technology File System (NTFS)
Eliminate fixed-size short names, Implement a more thorough permissions scheme, Provide good performance, Support large files, Provide extra functionality:
- **Master File Table –** sequence of 1KB records, similar to inodes but more flexible

RAID – Redundant Arrays of Inexpensive Disks (spread work across several disks)

Deadlock

Reusable resources – can be used by one process at a time, released and used by another process (printers, memory, locks, semaphores, processors, etc.)
Consumable resources – dynamically created and destroyed, can only be allocated once (interrupts, signals, messages)
Deadlock – the permanent blocking of a set of processes that either compete for system resources or communicate w/each other.

- Each process in the set is blocked
- Waiting for an event that can only be caused by another process in the set
- Resources are finite
- Processes wait if a resource they need is unavailable
- Resources may be held by other waiting processes

Necessary Conditions – these conditions must exist for deadlock to happen

- **Mutual exclusion –** only one process may use a resource at a time
- **Hold and Wait –** a process may hold allocated resources while awaiting assignment of other resources
- **No preemption –** a resource cannot be forcibly removed from a process holding it
- **Circular wait –** a closed chain of processes such that each process holds at least one resource needed by next

Together these form the **necessary** and **sufficient** conditions for deadlock

---

Solutions: Prevention, Avoidance, Detection & Recovery, Do Nothing
Deadlock Prevention – ensure one of the four conditions doesn't occur.

- **Break** mutual exclusion – not much help because it is often required for correctness
- **Break** hold and wait – processes must request all resources at once, and will block until can grab all of them
  - May end up waiting a long time
  - May hold resources for way too long
  - May not know all resources beforehand
- **Break** no pre-emption
  - Need to save the state of the process
  - May need to rollback
  - Impossible for consumable resources
- **Preventing** circular-wait – assign linear ordering to resource types and R can only request resources following R
  - Hard to determine total order when there are lots of resource types

Deadlock Avoidance – allows first three conditions but orders events to ensure **circular wait does not occur**. Requires knowledge of future resource usage and decides what order to choose

- Do not start a process if max resource requirements + max needs of all process running exceed total resources
- Do not grand an individual resource request if it might lead to deadlock

Safe States – a state is safe if there is at least one sequence of process executions that does not lead to deadlock.
Deadlock avoidance algorithm

- For every resource request
- Update state assuming request is granted
- Check if new state is safe
- If so, continue
- If not, restore the old state and block the process until it is safe to grant the request

Avoidance problems – max resource requirements for each process must be known, processes must be independent, and there must be a fixed # of resources to allocate
Prevention & Avoidance is awkward and costly. Instead, we allow them to occur and detect & break it. Check for circular.
Detection is often used instead. We draw **Resource Allocation Graphs** and check for cycles in the RAGs.

- Nodes are processes
- Arcs from a resource to a process represent allocations
- Arcs from a process to a resource represents ungranted requests
- In systems with multiple instances of a resource, cycles don't guarantee circular waits (but could!)

Deadlock Recovery:

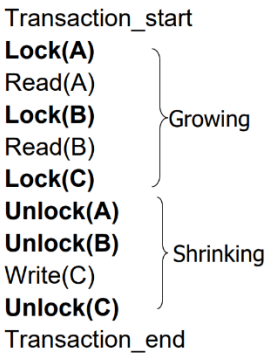
- **Drastic –** kill all deadlocked processes
- **Painful –** back up and restart deadlocked processes (non-determinism means night not deadlock this time)
- **Better –** selectively kill all deadlocked processes until cycle is broken (rerun detect after each kill)
- **Tricky –** selectively preempt resources until cycle is broken (processes must be rolled back)

Ostrich Algorithm – ignore the problem and hope it doesn't happen often. This works because modern OS **virtualize** most of the physical resources, eliminating the finite resources problem.
Transaction – a collection of operations that perform single logical function and executed atomically
Committed – a transaction that has completed successfully **Aborted –** a transaction that did not complete normally
Ensure atomicity – we write operations to a log on stable storage, then execute, then redo/undo following the log
Write ahead logging – before performing any operations, write intended operations to the log. They identify **old value, new value, data item.**

- Time consuming
- Large amount of space required
- Performance penalty
- **Checkpoints** help with first two problems

Concurrent Transactions – transactions must appear to execute in some arbitrary but serial order
- **Solution 1:** all transactions execute in critical section with a single common lock to protect shared data
- **Solution 2:** allow operations from multiple transactions to overlap as long as they don't conflict
  - **Conflicting operations –** if two transactions access same data item and at least one is a write
- **Conflict Serializable –** if a serial schedule can be obtained by swapping non-conflicting operations

Two Phase locking – individual data items have their own locks, **growing** and **shrinking** phases.
- **Growing** a transaction may obtain locks but may not release any locks
- **Shrinking** a transaction may release locks but may not acquire any new locks



Timestamp Protocols – each transaction gets unique timestamps before execution
WTS – largest timestamp of any transaction that wrote to the item
RTS – the largest timestamp of any transaction that read from the item
READS: If the transaction has earlier timestamp then WTS then the transaction needs to read a value that was already overwritten. **Abort and restart with new timestamp.**
WRITES: If the transaction has an earlier timestamp than RTS then the value produced by this write should have already been overwritten.

Set of Threads are in **Deadlock** when every process in the set is waiting for event that can only be caused by another process in the set.
Starvation – if it is waiting indefinitely because other threads are in some way preferred.

Security

Four Principles:
- **Confidentiality –** preventing unauthorized release of info
- **Integrity –** preventing unauthorized modification of info
- **Availability –** ensuring access to legitimate users
- **Authenticity –** verifying the identity of a user

Cryptography – techniques for communicating in the presence of adversaries
Interception – attacker gains access to info they should not have access to (loss of confidentiality)
Modification – attacker alters existing files, programs, etc. (loss of integrity)
Theft of Service – attacker installs daemon (attack on availability)
Fabrication – attacker creates counterfeit objects which appears from trusted source (attack on authenticity)
Intrusion detection – want to recognize when a system has been compromised
Signature based detection – examine system activity or network traffic for patterns that match known attacks
Anomaly detection – identify patterns of normal behavior and detect deviations from these patterns
Malware: malicious software
- **Trap Doors –** programs that contain secret entry points to allow attackers to bypass security
- **Logic bombs –** destructive code in legitimate program triggered by some event
- **Trojan horses** apparently useful program that tricks users into running it
- **Viruses** a program that can infect other programs by copying itself into them
- **Worms** are programs that spreads via network connections, do not need to attach to other programs

Stack & Buffer Overflow Attacks – overflow stack-allocated input buffer, return address with address of exploit code, overwrite next space in stack with exploit code itself.
Trusted Computing Base (TCB) – trust key things with your data (keyboard, computer, OS, app, etc). This is that set
Networking Attacks – passive (eavesdropping, monitor comms), vs **active** (modification/tampering with comm stream)
- **(P) Eavesdropping –** reading network packets, **defense:** obfuscate message contents (encrypt it)
  - **Same key cryptography –** fast enc/dec but distributing shared secret key is hard
  - **Public key crypto –** encryption key can be published, decryption is secret. 1000x slower though
- **(P) Traffic Analysis –** infer info based on sender/recipient, size, frequency, etc. **defense –** make all msg same length, all communications have same number of messages, etc.
- **(A) Replay –** capture legit message, resent → unauthorized message **defense –** nonce item so no reuse
- **(A) Modification –** alter contents of message to change effect **defense –** include message digest to confirm
- **(A) Masquerade –** pretends to be another user

Digest – fixed length string generated from content
Signature – provide a way to verify origin and integrity of a message (public key crypto can help)
Denial of Service Attack (DOS) – overload server with a lot of requests so there's no resources for legitimate ones (DDOS = distributed)
Principle of Least Privilege – figure out exactly what capabilities a program needs to run and grant It only those privs
Protection Domains – a set of objects and rights, assign each object a set of rights. These create protection domains/matrices
- Too costly to store such a sparse matrix
- Use an **access control list** (each object has ordered list, containing all domains that may access the object)

3 Waves of Security Attacks
- 1st wave – physical on wires/hardware
- 2nd wave – syntactic attacks on crypto/systems
- 3rd wave – semantic attacks, humans/computers trust info that they shouldn't