C

- malloc is used to *m*anually *alloc*ate a (contigious) block of memory on the heap
- calloc — malloc with initialization (typically to zeros)
- realloc reallocates to change the size of the block
- free releases the memory
- To put a line from stdin into the variable v: `scanf(“%x”, &v);` where x is a format specifier e.g. d for an integer
- `t* x` creates a pointer x to a variable of type t
- `*x` without a preceding type dereferences x
- The memory address of a variable can be obtained by prefixing the variable name with a & and can then be stored in a pointer
- `p += i` will add i*sizeof(type) to p where type is the type of the variable p is a pointer to
- void* can be cast to any t*

OS

Kernel

Dual mode operation

- Kernel space = the memory used by the kernel
- User space = the memory not used by the kernel (i.e. the memory used by application programs and any drivers etc. that are technically outside the kernel)
- To ensure security and stability, modern OSes do not allow programs running in user space to access kernel space
  - When a user process needs a privileged operation to be carried out, it uses a system call to get the kernel to do it
- **Hardware checks the mode bit** (0 = kernel mode, 1 = user mode) **before carrying out a privileged operation** and the OS will reset it to user mode once it finishes the privileged operation — mode bit is stored in a special CPU register
  - *This has been extended to form the protection rings of modern OSes used to allow drivers greater access than application software but less access than the kernel itself*

Kernel architectures

- Early OSes had an **unstructured monolithic kernel**
  - **Hard to debug**
  - **Little overhead in system calls**
- **Better monolithic kernels use a layered architecture** to add structure
  - **Improves development, debugging, and maintainability through separation of concerns and standardisation**
  - **Adds overhead as system calls must travel via intermediate layers to reach where they can be acted on**
  - Have to decide how many layers to have and what should go where — difficulties in doing this in a future-proof way is why monolithic kernels are no longer used
- **A microkernel moves as much code as possible out of the kernel into userspace programs**
  - Kernel provides processor and memory management, and interprocess communication
  - Filesystem operations, IO drivers etc. are userspace programs
  - **Application programs still access all the functionality through the kernel but the kernel acts as a proxy for the userspace extensions where relevant — adds overhead**
  - **Easy to extend and adapt**
- **Current OSes** are microkernel-esque but in the form of **loadable kernel modules (kernelspace programs which are only loaded in as required)**
  - **Removes the overhead as it is part of the kernel**

Processes

Memory

- **From the perspective of the process it has a contiguous block of memory (it's virtual address space) starting at address 0** but it starts at some point in physical address space and may be fragmented — OS performs the conversions
- **Structure of virtual address space (in order of ascending address):**
  - **Text = instructions**
  - **Data = global variables**
  - **Heap = dynamically allocated memory** (malloc and friends) **— grows up**
  - **Stack = local variables and function metadata** e.g. return address **— starts at maximum address and grows down**
  - The empty space between the stack and heap is what enables them to vary in size during execution

Process states

- **Process states:**
  - **new** — The process is being created
  - **ready** — The process is waiting to be assigned to a processor (CPU burst)
  - **running** — Instructions are being executed
  - **waiting** — The process is waiting for some event to occur (IO burst)
  - **terminated** — The process has finished execution

PCB

- To facilitate pre-emption, **the OS stores (in kernelspace) a process control block (PCB) for every process:**
- Process state
- Process ID
- Program counter value
- Other **CPU registers (written/read when an interrupt occurs/finishes)** e.g. pre-emptive context switching
- CPU scheduling information
- Memory management information
- IO management information
- Accounting information — CPU time used, wall time elapsed since start etc.
- *The PCBs are stored in an array called the process table*

Lifecycle: Creation

- **Each user process is created by a system process** — if a user process wants to create a process it does so via a system call (fork in *nix)
- **fork copies the address space of the parent into the address space of the child** — child carries on executing the same text (instructions) with the same values in variables (but copies, so changes made don't affect others) from the same point (program counter)
  - It will then continue into an if block that tells it what to do if its a child (**from the perspective of the child the fork call returned 0 but from the perspective of the parent it returned the PID of the child**)
    - execlp <u>overwrites</u> the current text (and variables) with some executable file

Lifecycle: Termination

- **When a process terminates it returns an exit code to the parent and the operating system frees all the resources allocated to the process**
- **The wait system call is called by a parent to block until and then receive the exit code of a child**
- **Operating system frees resources when the child calls exit but does not remove the process' entry in the process table** (releasing the PID back into the pool) **until the parent receives the exit code**
    - **While waiting** (possibly until the parent orphans it e.g. reboot) **for the exit code to be received, the process is called a <u>zombie</u> process**
    - *The process of receiving the exit code is called reaping*
- **If the parent exits before the child** (e.g. closing a terminal window that is still running a nohup'd process)**, the child becomes an <u>orphan</u> process**
    - In *nix, the init process (PID=1) will adopt the orphan process and wait on it
- A parent can kill its child using the kill system call

Interprocess communication

- **Shared memory = kernel allocates some memory to both processes and they then handle things (including synchronisation) themselves**
  - More specifically, a subset of the address space of one process is made accessible to the other
  - If one process is only a producer and the other is only a consumer, there are minimal synchronisation issues
    - Unix ordinary pipes are essentially shared memory where the process on the left is purely a producer and the process on the right is purely a consumer
    - This is actually fairly easy to have by allowing each to switch role frequently but only at the same time as the other
- **Message passing = processes send messages to each other using system calls** e.g. send and receive
  - Can be synchronous (e.g. block until the message we sent is received) or asynchronous (e.g. get the message if there is one, else return null instead of waiting for one to arrive)
  - Each process has a “mailbox” that its messages are sent to and received from
  - Unix named pipes are essentially message passing

Scheduling

Metrics and terminology

- **CPU utilisation = (time CPU is busy)/(time ready queue is non-empty)**
- **Throughput = number of processes that complete per unit time**
- **Turnaround time = amount of time to complete a process**
- **Waiting time = amount of time a process spends in the ready queue** = turnaround time − burst length
- **Response time = time elapsed between a process submitting its request and running for the first time**
  - Allows us to evaluate processes we don’t expect to end
- Non-preemptive scheduling = Once a process has the CPU, it keeps it <u>until the process switches from CPU burst to IO burst</u> (moves into waiting state)
- Preemptive scheduling = The execution of a process can be interrupted during its CPU burst (moves back to ready state) to give the CPU to another process

Queues

- Each queue consists of PCBs which store a pointer to the next PCB in the queue
- The short term scheduler selects the next process to be executed from the ready queue (the set of all processes in the ready state)
- The long term scheduler selects the next process to be executed from the job queue (the set of all processes in the new state)
    - Not present in Linux or Windows — short term scheduler also deals with new processes
- Device drivers select the next process to access a device from the relevant device queue (the set of all processes waiting on that device)

Algorithms

- **First come first-serve** (FCFS)
  - Non-preemptive
- **Shortest-job first** (SJF): Job in ready queue with shortest next CPU burst is selected, tie-break with FCFS
  - Can be pre-emptive (switch if a process with shorter time than the remaining time of the current job is added to queue) or non-preemptive
  - **Assuming CPU burst time estimates are sufficiently accurate, preemptive SJF is the algorithm that gives minimum average waiting time** and non-preemptive SJF has the minimal average waiting time over non-preemptive algorithms
  - To estimate CPU burst times exponential moving average is used: **Let t_i = actual time of burst i and τ_i = estimated time of burst i. $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$, for some constant $\alpha \in [0, 1]$.** That is τ_{n+1} = (Σ from i=0 to i=n-1 of $\alpha(1-a)^i t_{n-i}$) + $(1 - \alpha)^{n+1}\tau_0$
- **Priority scheduling — generalization of SJF,** each process in the queue is assigned a priority and CPU is allocated to highest (which depending on OS may be the numerically smallest) priority
  - **To avoid starvation, use aging — increase priority of processes as they spend time in the ready queue**
- **Round robin**: Each process is dequeued, ran for a small time quantum q, then pre-empted and enqueued to back of queue
  - For large q, approaches FCFS which is not good
  - For small q, we have poor utilization due to overhead of frequent context switching
  - **Best response time of the algorithms we have seen**, just have to make sure to get q right

Memory

Misc

- Memory protection = ensuring that each process can only access the address space belonging to it
- The MMU (memory management unit) is a specialised piece of hardware (typically built into CPU) that translates the virtual addresses (from logical address space) of the CPU into the physical addresses (from physical address space) of the memory bus
- **Internal fragmentation = a process has been allocated more memory than it requested**
- **External fragmentation = there is enough free memory to fulfil a request but it is not contiguous**

Contiguous memory allocation

- In contiguous memory allocation, the only difference of physical address space to logical address space is all addresses are offset by some base address
- **Relocation (base address) and limit (block size) for the scheduled process are loaded into the MMU's registers by the OS — MMU checks the logical address against the limit <u>then</u> if valid adds the base address**
- A hole = a section of contiguous memory that is free
  - The more holes there are, the more memory the OS needs to keep track of them (as the length and start of each hole is stored) regardless of the allocation technique, and the less likely contiguous allocation is to be successful
- Strategies:
  - **Fixed-size** = split memory up into a fixed number of blocks called partitions, give each process enough (contiguous) partitions for it to have enough memory — each chunk is contiguous and they are in relation to each other (all come from the same hole)
  - **First-fit** = allocate the first hole that is big enough
  - **Best-fit** = allocate the smallest hole that is big enough
  - **Worst-fit** = allocate the largest hole
- **Compaction = rearrangement of memory to combine holes** to form a single large hole — think hard drive defragging

Non-contiguous memory allocation: Segmentation

- **Segmentation splits memory into <u>variable-size</u> blocks called segments and allocates each process one or more segments** — each segment is contiguous but they may be non-contiguous in relation to each other
- **Each logical address now consists not only of an offset from a base address but also a segment number**
- **MMU stores the segment table (an array indexed by segment number) in which each element is a pair (base address, limit)**
- **Upon receiving an address, the MMU splits it into segment number and offset, looks up the segment number in the segment table, then proceeds as it did in contiguous memory allocation**
- External fragmentation can still occur but is much less frequent than with contiguous as holes can be much smaller (segment size)

Non-contiguous memory allocation: Paging

- **Paging divides the *program* into fixed-size blocks called *pages* and memory into fixed-size blocks called frames**
- **Page numbers are mapped to frame numbers (base addresses) by the page table**
  - **The OS maintains a separate page table for each process**
- **Logical addresses are made of a page number and a page offset**
- **OS uses page number to find frame number (base address in contiguous terms) in page table and puts it and page offset (logical address in contiguous terms) into MMU** — *a limit can be stored with the page table but this is not commonplace as there are other memory protection mechanisms in use*
  - Unlike segmentation, the table is not stored in the MMU but instead in main memory
- Number of bits required for page offset = $\log_2$(page size in bytes)
- **Number of bits used for page numbers** (note this may be more than is needed to be able to address the number of pages in a given system depending how much RAM it has) **= number of bits in addresses − number of bits required for page offset**
- **Number of entries in a conventional page table = $2^{\text{number of bits used for page numbers}}$**

Page tables: Lookups

- Storing page tables in main memory begs the question of how to address the page table: **When a process is scheduled the base address of its page table is loaded into PTBR (page table base register) from its PCB**
  - **Each page is looked up by adding page number to base address and accessing that memory address to find the frame number** which is then loaded into MMU alongside the page offset — page tables are arrays (and so contiguous)
  - **Each fetch requires two memory accesses** which is inefficient
- **Translation lookaside buffer (TLB) is a CPU component which caches** recently/frequently used **page table entries**
  - Logical address is checked against all entries in TLB in parallel to find a match (takes negilbilbe (less than 1 CPU cycle) time) and the previously described 2 * main memory accesses procedure is used in the case of a cache miss
    - **After a cache miss is resolved, it is moved into the cache**, using some eviction strategy if TLB is full
    - **Effective access time = main memory access time * (hit ratio + (1 − hit ratio) * 2)**
- **To provide memory protection, if the same TLB is storing page tables for multiple processes, each entry has an address space identifier (ASID) to identify the process**
  - **If found in TLB but ASIDs don't match, is a cache miss**

Page tables: Optimizations

- As each process has its own page table and each page table covers at least the entirety of memory, **most entries in a page table are not used but are occupying space in main memory**
- **Hierarchical paging: Page table is itself split into pages called inner page tables and the outer page table is used to store the mappings into the inner page tables**
  - **Outer page table is complete but <u>each inner page table is only stored if not empty</u>**
  - **Each logical address consists of outer page number, inner page number, and offset** instead of page number and offset
    - We are trading off time for space as we now require 3 memory accesses in the case of cache misses instead of 2
  - **TLB works using the concatenation of the outer and inner page numbers**
  - Can have any number of layers by repeatedly applying this processes
- *Current 64-bit OSes (and CPUs) actually only support 48 bit address space as the depth of the hierarchical page tables becomes prohibitive, a proposed solution is* **hashed page table: A hashmap (with a fixed number of linked list buckets storing (page number, frame number) pairs) keyed by page number**
- PowerPC used an **inverted page table to allow only one page table to be used for all processes: Entries are in order of ascending frame number and each entry contains a page number and pid (process id). Page table is linearly scanned to find entry with the matching page number and pid, number of comparisons = frame number**
  - Note we now only store the pages that actually exist — number of entries = amount of RAM / page size
  - Linear scanning is bad compared to the constant time access of regular page tables but does save on space — PowerPC is dead but this principle will come back to an extent if hashed page tables begin to be used

Threads

Threads

- **Unlike** (forking) **processes, threads share code, global variables, heaps, and resource handles**
    - **Still have their own stack (local variables) and registers**
    - Shared global variables make it possible to have inter-thread communication without having to go via OS
- Thread creation is faster than process creation and thread switching is faster than context (process) switching
- **Amdahl's law: Speed-up from parallelising is upper bounded by 1/(S+(1−S)/N) where S is the proportion of time spent running serial (non-parallelizable) code and N is number of cores**
    - Is only an upper bound — parallelizing sequential code adds overhead from synchronization primitives etc. and the effect of using more cores for parallel code can be limited by memory bandwidth, cache capacities etc.

Threading strategies

- **One-to-one model: Each user level thread has its own kernel level thread**
  - **Threads can run fully in parallel**
  - **Kernel can see all the threads in the user process and hence thread management** (synchronisation and scheduling) **can be outsourced to kernel**
  - **Have to ask kernel before creating threads** — adds overhead and it may say this process has exceeded its quota
- **Many-to-one model: Each user level thread in the same process has the same kernel level thread**
  - **Only one thread can run at once**
  - **Thread management is done by the user process which requires additional programming but is more flexible** — the kernel doesn't even know the code is multi-threaded
- **Many-to-many model: Requests more than one kernel thread** (in accordance with desired degree of parallelization) as in one-to-one **but manages the allocation of** (potentially more than the number of kernel threads) **user level threads onto the kernel level threads itself** as in many-to-one

Signals

- **Synchronous (caused by something inside the process) signals** (e.g. divide by zero error, seg fault) **are delivered only to the thread that carried out the operation that caused it**
- **Asynchronous (caused by something unrelated to what the process was doing) signals** (e.g. SIGINT) **are delivered to all threads**

Synchronization

Terminology

- From now on we use thread and process interchangeably
- **Critical section = a section of code in which shared resources** (such as global variables of threads) **are accessed**
- **Mutual exclusion = at any given moment the number of processes that are in any given critical section is at most one**
- **Progress = no deadlock, if at least one process wants to enter a critical section and no process is currently in that critical section, one of the waiting processes will be allowed to enter the critical section**
- **Bounded waiting = no starvation, every process that ever waits to enter a critical section will eventually be able to do so**
    - **Bounded waiting ⇒ progress** *(for this module, this is not true under some definitions)*
- **Critical section problem: Find a protocol that provides mutual exclusion and bounded waiting (and so also progress)**

Peterson’s Algorithm

- **Peterson’s algorithm for synchronisation of 2 threads:**
    - **Use a shared variable int turn to denote which (0 or 1) thread ID’s turn it is. Use a shared array boolean flag[2] indeed by thread ID to denote whether each thread <u>wishes to enter or is currently in</u> critical section**
    - **If a thead wishes to enter critical section it sets its flag to true <u>then</u> sets turn to the thread ID of the <u>other</u> thread then busy waits until it is its turn <u>or the other’s flag is false</u>**
    - **Upon leaving critical section, sets its flag to false (<u>does not change turn</u>**)
    - *This has been extended to more than 2 variables since its initial publication*
- Peterson’s algorithm is a solution to the critical section problem, however:
    - Busy wait is inefficient
    - Modern CPUs rearrange operations (in a way that would give the same final result for a single threaded program e.g. change turn before changing element in flag) for efficiency which breaks the correctness of Peterson’s algorithm
- **Modern CPUs provide certain operations that are guaranteed to be atomic**, this has made Peterson’s obsolete
    - boolean test_and_set(boolean* target) sets target to true and returns the value of target when the operation was called — implemented atomically by the hardware

Modern barebones

```
// Code for thread i
do {
    waiting[i] = true;
    lock_was_locked = true;
    while (waiting[i] && lock_was_locked) {
        lock_was_locked = test_and_set(&lock);}
    waiting[i] = false;

    // critical section here

    // Give way to the right to provide bounded waiting
    j = (i + 1) % n;
    // Enumerate processes until we find one that is waiting or have checked them all
    while (!waiting[j] && j != i) {
        j = (j + 1) % n;}
    waiting[j] = false;
    if (j == i) {
        // If no waiting processes, release lock
        lock = false;}
} while (true);
```

OS primitives

- Atomic hardware instructions are used by OS designers to provide synchronization primitives such as **mutex locks and semaphores** to avoid programmers having to write the lengthy code on the previous slide themself
- **Unlike true/false mutex locks, semaphores are integers (0 iff unavailable, positive iff available)**
- A semaphore initialized to m allows up to m threads to access the critical section

OS primitives: Pseudocodes

- Each of these functions is implemented atomically

-
  ```
  void lock(boolean* mutex) {
      if (*mutex) {
          waiting_processes.append(self);
          sleep(self);}
      *mutex = True;}
  ```

-
  ```
  void unlock(boolean* mutex) {
      *mutex = False;
      if (waiting_processes.length > 0) {
          Process p = random.choice(waiting_processes);
          resume(p);}}
  ```

-
  ```
  void wait(int* semaphore) {
      if (*semaphore == 0) {
          waiting_proesses.append(self);
          sleep(self);}
      *semaphore -= 1;}
  ```

-
  ```
  void signal(int* semaphore) {
      *semaphore += 1;
      if (waiting_processes.length > 0) {
          Process p = random.choice(waiting_processes);
          resume(p);}}
  ```

Evaluation: Metrics

- Potential issues with mutual exclusion protocols:
  - Deadlock
  - **Starvation = There exists a process that has to wait indefinitely to enter its critical section, that is there is not bounded waiting — this is why we use randomness to decide which thread to resume**
  - **Priority inversion = Lower priority process holds a lock needed by a higher priority process**. Contrary to what their priorities suggest, the lower priority process should be scheduled more frequently than the higher priority process
    - **Priority inheritance protocol: If priority of waiting process is higher than that of blocking process, give priority of waiting process to blocking process until it releases the lock**

Evaluation: Test cases

- Bounded buffer problem: n buffers, each with capacity 1. One producer and one consumer. Stop producer from producing iff all buffers are full. Stop consumer from consuming iff all buffers are empty. Don't allow producer and consumer to access the buffers at the same time.
    - Solution: Use a mutex to control access to the buffers. Use two semaphores to count number of full (initialised to 0) and empty (initialised to n) buffers.
- *(First)* Readers and writers problem: A data set (of unbounded capacity) is shared between readers and writers. Allow multiple readers to access at once but only allow one writer to access at once (and only if no readers are accessing), that is a thread can access iff a writer is not currently accessing. Give priority to readers, this risks starvation of writers but has been deemed acceptable.
    - Solution: Use a mutex to control access to a count of how many are currently reading. Use a mutex to control access to the data set by the writer (and reader if no one is is currently reading).
- Dining philosophers problem: n philosophers are sat around a circular table with a bowel each. To eat they need to hold 2 chopsticks. Philosophers are happy to think when not eating for as long as necessary but do not wish to starve. There is one chopstick in between each pair of bowls (so there are n chopsticks in total), that is that a philosopher can eat iff their neighbours are both thinking.
    - **Solution: Use an array of mutexes to control each chopstick. Philosophers with odd index pick up their left chopstick first, philosophers with even index pick up their right chopstick first. Pick up (lock) both chopsticks before eating. Put down (unlock) both chopsticks after eating.**
        - **If we allowed every philosopher to pick up their left chopstick first, deadlock would be possible**

Deadlock

- **A set of processes is in deadlock if each process in the set is waiting for an event** (usually release of a resource) **that can only be caused by another process in the set** and so none of the events can ever occur e.g. each two threads each holds one mutex and each requires both mutexes to enter critical section
- We can model a system as a set of resource types $\{R_1, \ldots, R_m\}$ each R_i of which has a count W_i of its number of instances, and a set of processes $\{P_1, \ldots, P_n\}$
- **<u>Necessary but not sufficient</u> conditions for deadlock:**
  - **Mutual exclusion** — The resource instances in question are subject to mutual exclusion enforcement
  - **Hold and wait** — There is a process that holds a resource and is waiting to acquire additional resources
  - **No preemption** — The resources being waited on can only be released by the processes holding them and they will only do so once they have finished using them
  - **Circular wait** — There exists a subset $\{P_1', \ldots, P_m'\}$ of $\{P_1, \ldots, P_n\}$ s.t. P_1' is waiting for P_2', P_2' is waiting for P_3', …, P_{m-1}' is waiting for P_m', P_m' is waiting for P_0 — *this is sufficient if there is only one instance of every resource involved*
- **A resource allocation graph is a directed graph $G = (V, E)$ where $V = \{P_1, \ldots, P_n, R_1, \ldots, R_m\}$ and $E = \{(P_i, R_j)$ s.t. P_i is waiting for $R_j\} \cup \{(R_j, P_i)$ s.t. P_i holds $R_j\}$**
  - **We draw processes as circular nodes and resources as square nodes with a dot inside for each instance**
    - **Request edges are to the box**
    - **Holding edges are from a particular dot**
  - **Cyclic ⇔ circular wait** ⇐ deadlock — **acyclic ⇒ no deadlock** but cannot definitively conclude if cyclic *(unless only one instance of each resource)*

Deadlock detection algorithm

- This seems like the banker's safety algorithm written nicer but they were treated separately in lectures?

```
1.  Create a table with a row for each process and columns for how many
    instances of each resource are currently allocated to and requested by
    (in separate columns) the process in each row
2.  Create an array for how many currently available instances of each
    resource there are
3.  Create an array of booleans finish for each process
4.  While true
    a.  Find a process P with finish == false whose resource request can be
        entirely satisfied by what is currently available
    b.  If there is no such process
        i.  return "deadlock"
    c.  Make all the resources held by P available and change P's finish to
        true
    d.  If all finishes are true
        i.  return "no deadlock"
```

Deadlock prevention

- **To guarantee we have prevented deadlock we only need to guarantee that any one of the necessary conditions does not hold**
  - As the conditions we are falsifying are necessary but not sufficient, we will sometimes be reducing parallelism needlessly
- Mutual exclusion generally is non-negotiable
- We can prevent hold and wait by not allowing a process to request resource(s) if it currently holds any resources (and granting resources atomically and only when the entire request can be satisfied)
- We can prevent no preemption by forcing processes to give up all their currently held resources if they are waiting for a resource request to be fulfilled
- **We can prevent circular wait by imposing an ordering on resources and requiring processes to request resources in increasing order and only request resources greater than those it already holds**
  - So again **a process that could otherwise be risking causing deadlock must give up its currently held resources and wait until its entire request can be satisfied to get them back**

Deadlock avoidance

- Deadlock avoidance is more context-specific (knows upper bounds on future requests) than deadlock prevention which allows it to avoid refusing requests unnecessarily
- **A system state is a safe state iff there exists no possible sequence of future requests that when applied to that state will cause deadlock**
- **Deadlock avoidance algorithm: Each process declares the maximum number of instances of each resource type it could ever need. Requests are granted if the state immediately after granting it would be a safe state and there is currently sufficient resources available to make the grant, otherwise the requesting process is made to wait until the state is such that these conditions are satisfied.**
- **Banker's safety algorithm (state safety checking given known needs):**

```
1.  requests = [(P, R) for P in processes for R in resources if R in P.requests]
2.  requests_data = [(P, R, MAX, ALLOCATED) for (P, R) in requests]
3.  resource_data = [(R, AVAILABLE) for R in resources]
4.  while True
    a.  needs = [(P, R, NEED = MAX - ALLOCATED) for (P, R, MAX, ALLOCATED) in
        requests]
    b.  Find a process P whose NEEDs does not exceed the corresponding AVAILABLEs
    c.  If there is no such P then return "Unsafe"
    d.  Add P's ALLOCATEDs to the corresponding AVAILABLEs, then remove P
    e.  If there are no processes left then return "Safe"
```

Interpreting banker’s algorithm questions

- “A request arrives for … can this be granted immediately?” means: Move those resources from available to allocated (note it certainly cannot be granted immediately if the request exceeds the available and it doesn’t fit in our model if the request exceeds the <u>need</u>) and run bankers algorithm (remembering to update need to reflect the change to allocated) to check if the state produced is a safe state
  - This is called the “Resource Request Algorithm”
- “Processes … request resources …, can these be granted together or individually” means: (To the best of my knowledge, but I have been unable to verify) check the above separately for individually (and take conjunction of (is safe)) and do the above for all the requests at the same time for together

Computer Networks

Network attacks

- **A connection is called half-open when the server has received the client's SYN and done the server side processing it needs to create a connection (so is about to send SYN ACK)**
- **A connection is called established when the server has received the client's ACK**
- **SYN flood attack: Sends many SYN packets** (with many different source ips (as can put whatever information you like in a packet)) to create many half-open connections **and never ACK the SYN ACK** (never establish the connection)
    - **Server expends resources on creating the connections and these resources will be held until it decides the handshake has timed out**
    - When legitimate users try to connect they will be unable to as the server will not have sufficient resources to reply before the user's computer times out the request
- **A router's routing algorithm determines which IP address that it has a connection to would be most useful to send the packet to based on its destination IP and ARP is then used to convert this IP into a MAC address to give to the link layer**
    - ARP cache poisoning: ARP requests are broadcasts, and ARP supports sendings replies without requests to preemptively update

Layers

- **Application** — Responsible for application specific high-level data presentation, does not consider data transmission
- **Transport** — Responsible for splitting the message from the application layer into chunks suitable for transmission over an end-to-end connection between two devices
  - **Splits data into segments**
  - **Adds destination and source port number, and checksums for each segment**
- **Network** — Turns the chunks into packets suitable for transmission over a hop-to-hop connection to allow the message to be routed to a recipient located outside the LAN
  - Turns segments into packets — splits one segment into several packets if it's too big
  - **Adds source and destination IP** addresses to each packet
  - **Performs routing across subnets**
- **Link** — Prepares packets to be send from one device to another (one hop)
  - **Adds source and destination MAC** addresses if ethernet
    - **MAC addresses change on each hop whereas all other layers stay the same (but NAT breaks this principle by changing the IP address and port number at router)**
  - **Performs routing within subnets**
- **Physical** — Sends the packets, converts each packet into a physical phenomenon on a specific link
- Each layer adds its own header (and some add a trailer as well)
- Packets are built as layer 4 which is then placed inside layer 3 and then layer 2, then layer 1
  - The receiving device process and discards layer 1, then layer 2 and so on

Application: HTTP

- **Sockets are the API between the application layer controlled by the user process and the lower layers controlled by the operating system**
  - Application read/writes to a socket (which in *nix is a file) and OS magically turns that from/to data transfer over the network
- **Non-persistent HTTP (HTTP1.0) serves each object (file) over a new TCP connection whereas persistent HTTP (HTTP 1.1+) allows multiple objects to share a TCP connection** which is very useful for Web2.0+ as the HTML initially requested and sent will inevitably required images, javascript, css etc to be requested and sent
  - Recall that establishing a TCP connection takes 2 RTT and allows a request to be sent and a response to be received
  - Using a TCP connection to send a request and receive a response takes 1 RTT
  - Creating multiple TCP connections allows requests to be sent in parallel reducing how many RTTs of wall time it takes to load the webpage
    - **HTTP1.1** *states that* **pipelining** *may be used* **(sending multiple requests inside 1 RTT over the same TCP connection)** *but it is not widely used (the HTTP2 equivalent is though) but* **Arpan's solutions assume it is being used**

Transport layer

Protocols

- The operating system maintains a mapping from port numbers to PIDs
- TCP provides a connection-oriented reliable in-order data transfer service
    - **TCP handshake:**
        - **Client sends SYN**
        - **Server creates a socket**
        - **Server sends SYN ACK**
        - **Client sends ACK <u>(and some data if it wishes)</u>**
        - **[Data transmission and ACKs continue until the connection ends]**
        - <u>Although this is 3 packets, it is 2 RTTs</u> (as each round trip is a client packet and a server packet) e.g. when accessing a webpage: client sends SYN, server sends SYN-ACK is first RTT; client sends ACK and request, server sends ACK and response is second RTT and the client now has their data (and will ACK it and request any referenced objects)
- UDP provides a connectionless best-effort data transfer service
    - Can identify corrupt data (as does have checksums) but can't maintain data order (as doesn't have sequence numbers) and doesn't retransmit lost data
    - No handshake required
    - Headers are smaller than TCP

Reliable data transfer

- Reliable data transfer = retransmission of lost packets and the reordering of out of order packets
- **Link utilisation (U) = amount of data sent in a period of time/amount of data that could have been sent in that period of time**
- **Stop and wait ARQ (automatic repeat request): After each packet, sender waits to send the next packet until it has received the ACK. Retransmits the packet it is waiting for the ACK for if timeout is reached.**
  - If ACK gets lost or overly delayed, a packet may be retransmitted unnecessarily but sequence numbers allow the duplication to be detected
  - **If an out of order packet** (determined by sequence number) **or corrupt** (determined by checksum) **is received, it is discarded and not ACKed**
  - $U = L/(L + R*RTT)$ where L is packet size and R is bandwidth but **$L \ll R*RTT$ so utilization is poor**

Reliable data transfer: Pipeline protocols

- **We can send up to (buffer capacity of the receiver may limit us) R*RTT extra bytes of data while waiting for an ACK** if we have a protocol that allows us to be waiting on multiple ACKs at once
- **Go-back-N (GBN) ARQ: Sender can send up to N (the sender window) packets while waiting for ACK** — with each (non-duplicate) ACK the window slides along by 1, it isn't batches
  - Receiver maintains a local variable expectedSeq which is the sequence number of the next packet they expect to receive
  - **If the receiver receives packet i intact and i=expectedSeq, they send ACK i which acknowledges <u>all</u> packets up to and including i (cumulative ack) and increment expectedSeq**
    - When each side sends i=expectedSeq they will have previously sent i=expectedSeq-1 etc. but they may not have been received
  - **If the receiver receives packet i but i ≠ expectedSeq (out of order packet) or the packet is corrupt** (checksums don't mach), **they send ACK expectedSeq-1 and discard packet i**
    - **Sender will only retransmit expectedSeq once its timeout is up without receiving a suitable (greater than or equal to expectedSeq) ACK**, such an ACK would show that the packet was received ACK was just lost or delayed
  - Window size can be up to maximum sequence number

Reliable data transfer: Pipeline protocols: Selective receive

- **Selective-receive ARQ: GBN but there is a receive window of size n as well and out of order packets are not discarded** (and so packets sent while waiting for a loss to be detected don't need to be retransmitted) **as long as they fall inside the receive window (expectedSeq through to expectedSeq+n−1)**
    - **ACKs are individual instead of cumulative (sender will retransmit packet with seq i once its timeout is up if it has not received exactly ACK i)**
    - Window size (assuming windows are same size) can be up to half the maximum sequence number (as two windows need to be accommodated and in worst case won't overlap at all)

Reliable data transfer: TCP

- **TCP can be seen as selective-receive but with <u>cumulative acks (of new expectedSeq)</u>**
- **TCP increments seq and ack number by length of previous packet**, that is that they give the byte number of the first byte in the packet with relation to the whole session
- TCP supports duplex communication — **ACKs can carry data (and so have a seq num as well as an ack num)**
- Timeout is fairly long, so TCP has **fast retransmit. If 3 duplicate ACKs are received, retransmit without waiting for timeout**
- **Flow control = Sending rate is altered based on receivers download speed** to avoid the receiver running out of buffer space
  - **TCP packets have a receive window <u>field</u> (rwnd) which is used to advertise how much free buffer space the host has**
    - **Send window width of sender should not exceed rwnd of receiver**

TCP: Congestion control

- **We say a network is congested for us if a router we are using is receiving data faster than it can send data** (as it has to verify checksums, run routing algorithm etc.) **and so is having to queue packets in its buffer**
  - If buffer is not full but also not empty, packets will be delayed
    - The more congestion there is, the longer packets have to wait to reach the front of the queue
  - If buffer becomes full, packets will be lost
- **Congestion control = Sending rate is altered based on network congestion** so as to make the most of what capacity is available but not add to congestion
- **Sender maintains a <u>local</u> variable cwmd (congestion window size) and its send window width should not exceed this** (or more precisely should be not exceed the lower of this and rwnd)
- **TCP assumes a network to be congested if it has to retransmit** and treats timeout retransmits more seriously than fast retransmits
- **AIMD (additive increase, multiplicative decrease): Increment (by MSS (maximum segment size) cwnd <u>every RTT</u> if no loss, half cwnd if loss (retransmit required)**
  - **If the retransmit is timeout not fast** (packets appear to be being lost not simply delayed)**, resets to slow start (1MSS etc) with ssthresh of half of current cwmd**
  - Designed to only use our fair share of the network capacity — n AIMD (i.e. TCP) senders sharing a link will converge to use 1/n of the capacity each
- **Slow-start phase: At start of session,** increment (by MSS) cwnd <u>with every ACK</u> (i.e. **cwnd doubles every RTT**) **from a very small value** (e.g. MSS) **through to ssthresh, switching to AIMD once ssthresh is reached** (we don't overshoot)**. If there is a loss (retransmit required), halves ssthresh and cwnd and continue with slow start.**
  - "Slow" start does start slow but increases rapidly, gives a faster start overall by speeding up convergence

Network layer

Routing: Subnets

- **CIDR** (Classless Inter-Domain Routing) is a notation for subnet masks — **write a (ideally the first) IP in the block followed by a forward slash followed by the number of bits given to the network ID** *e.g. the IP ranges reserved for private IPs are 10.0.0.0/8 (10.0.0.0 – 10.255.255.255), 172.16.0.0/12 (172.16.0.0 – 172.31.255.255), and 192.168.0.0/16 (192.168.0.0 – 192.168.255.255)*
- **Network interfaces with IP addresses on the same subnet are connected to the same switch and so communicate with each other without the need for IP based routing** as switch uses MAC addresses to decide routing (link-layer routing)
  - ARP is used to convert the IP to a MAC
- **Network interfaces with IP addresses on different subnets are connected to different switches and so communicate with each other via a gateway router using IP based** (network layer) **routing** and will after enough hops reach the correct subnet at which point a switch provides the last mile delivery using MAC address
- NAT: Only gateway routers are given public (globally unique) addresses, within the subnet private addresses (unique within the subnet but not globally) are used and mapped to and from the public address of the gateway by the gateway
  - **Gateway has a NAT (network address translation) table — assigns a unique port number to each combination of private IP and port number that is currently in use**

Routing

- The telephone network uses **circuit switching which creates a fixed dedicated connection between two end points for the entirety of the data transmission** — provides fixed bandwidth, can be nice to know it's guaranteed
- The internet uses **packet switching to allow multiple devices to share a communication channel without conflicts** — not only makes better use of capacity but also allows data to change route mid-transmission if network conditions change
- It is important to remember that each network interface has its own IP address and so a router will have multiple IPs
- **Router looks up the destination IP of an incoming packet in its routing table to decide which of its interfaces it should be sent down**
    - To save on space, each entry is a range of IPs — longest prefix matching: **each entry is an (binary) IP prefix, the longest matching prefix is the one whose associated link is used**

Routing: Algorithms

- We can consider the problem of constructing a routing table as that of finding the least cost path to each other node from some node in a graph of routers with cost weighted edges between them representing links
- **Local routing algorithms only require each router to have knowledge of its neighbourhood e.g. Distance vector (DV) algorithm** (detailed below)
  - Let N(x) = neighbours of x
  - Dynamic programming: Let $d_x(y)$ = length of shortest path from x to y
  - Node x maintains a vector D_x of its estimate $D_x(y)$ of $d_x(y)$ $\forall y \in N(x)$
    - $\mathbf{D_x(y) = min\ \{c(x,v) + D_v(y)\ s.t.\ v \in N(x)\}}$ — note the similarities to Dijkstra
  - **Whenever a node's vector <u>changes</u> it sends it to all its neighbours** e.g. in the above x knows D_v because it was sent it by v
    - **A node update its vector when one of its link costs changes or it receives a new vector from one of its neighbours**
- **Global routing algorithms require each router to have knowledge of the neighbourhood of every router e.g. Dijkstra** (for once we do actually want the all the paths from the source)
  - Routers broadcast the current state (cost) of each of their links to all their neighbours which can repeat the message to their neighbours alongside their information and so on so everyone gets all the information
  - Used in open shortest path first protocol

Data transmission layers

Link/Physical

- **Store-and-forward principle of packet switching: A node can can only begin to forward a packet to the next node if it has received the entire packet** *(and hence has been able to verify integrity and drop the packet if the checksums don't match)*
  - Small packet size is important if you want low latency
- **Components of delay:**
  - **Propagation delay = distance of link / data propagation speed in medium** — physics limits how fast the information can travel
    - “Distance and speed limit of highway” together determine how long it takes each “lane” to travel
  - **Transmission time = (packet length)/(link bandwidth (throughput))**
    - Higher throughput gives more “lanes of highway”
  - **Processing time — parsing packet, verifying data integrity, and selecting output link**
  - **Queueing time — occurs when the rate of data into a node exceeds the bandwidth of its outgoing link**
    - **If buffer becomes full, packets will be dropped**