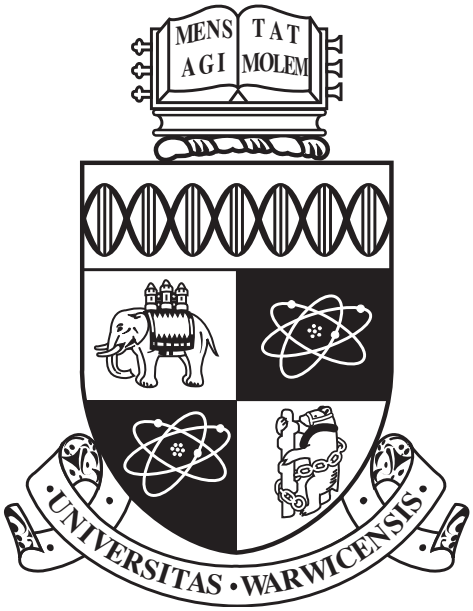
University of Warwick
Department of Computer Science

CS241

Operating Systems and Computer Networks



Cem Yilmaz
July 27, 2023

Contents

1 Operating Systems

1.1 Kernel

Definition 1.1. Kernel

Kernel is the core of an operating system and it is loaded into the memory at system start up. It is the process running at all times on the computer. There are certain functions only a kernel performs, such as memory management, process scheduling and file handling.

Kernel space is the part of the memory where the kernel executes. User space is the section of memory where processes run. Kernel space is kept protected from the user space and lastly, the kernel space can be accessed via user processes through system calls. These perform services like I/O operations or process creation.

Definition 1.2. Monolithic Kernel

The monolithic kernel contains all of the functionalities, meaning that it is a single layer kernel. Monolithic kernels make it difficult to debug the kernel code but are fast because there is little overhead in the system call interface.

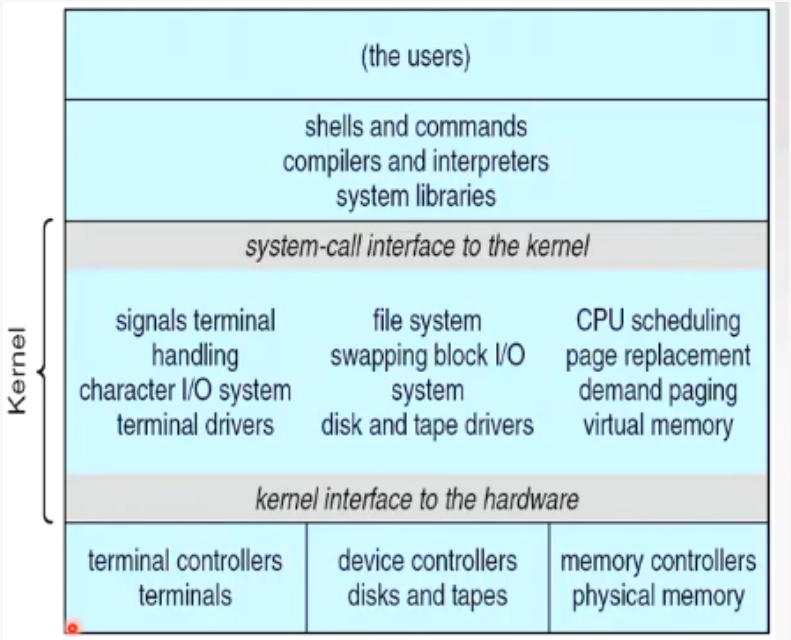


Figure 1: Monolithic Kernel

Early OS suffered because of hardware of their time. Early hardware did not support dual mode of operation. In MS-DOS the interfaces and levels of functionality were not well defined. To combat this people have introduced layers.

Definition 1.3. Layers

Dependencies between different parts of the kernel code can be reduced using a modular kernel design approach. Modular design can be achieved through separation of different layers. The bottom layer is the hardware and the top layer is the user. Layer K uses the services of layer $K-1$ and provides service to layer $K+1$ to distinguish.

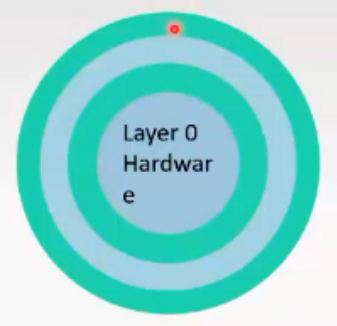


Figure 2: Layer N - User Interface

This allows developers to edit specific functionalities in specific layers only, as they're all independent.

Table 1: Layered Approach Pros and Cons

Pros	Cons
Simplicity of construction Ease of debugging Clear interface between layers	Defining layers is difficult Reduced efficiency due to number of calls to execute layers

Definition 1.4. Microkernel

The Mach operating system in mid 80s modularised the kernel using a microkernel approach. The idea is to remove all non-essential components from the kernel and implement them as either system and user-level programs. These resulted in a smaller kernel. Microkernels only provided minimal process and memory management inter-process communication.

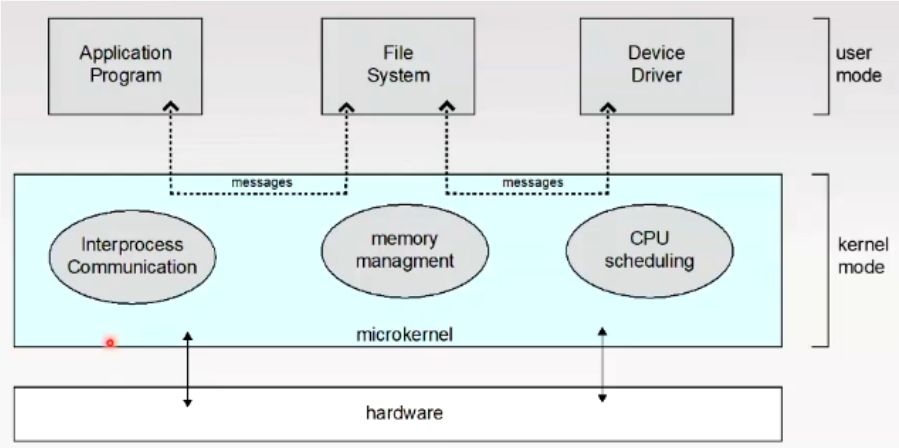


Figure 3: Microkernel

Table 2: Microkernel Pros and Cons

Pros	Cons
Extending the operating system is easy Security and reliability (most processes in userspace)	Efficiency - increased system call overhead

Definition 1.5. Loadable Kernel Module

The most modern approach to operating system design involves loadable kernel modules. Kernel provides core services while other services are implemented dynamically as the kernel is running.

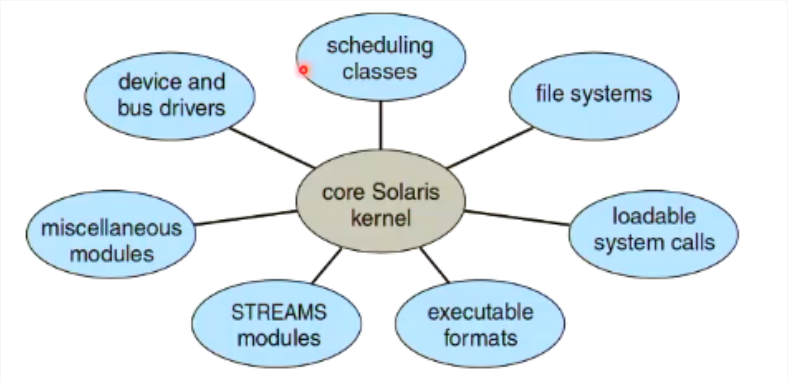


Figure 4: Loadable Kernel Modules

1.2 Processes

Definition 1.6. Process

A process is a program in execution. A program is passive entity stored on disk as an executable file. A process is active.

Example 1.1. Process in memory

We present processes in memory as a continuous box of space that can be taken by text (stores instructions), data (stores global variables), heap (dynamically allocated memory), stack (stores local variables and function parameters such as return address of a function). The space between stack and heap allow them to grow or shrink during a program run-time

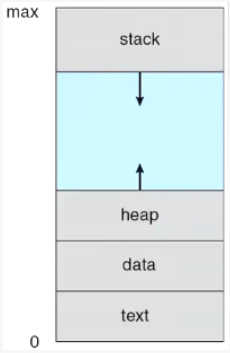


Figure 5: Virtual address space of a process

Definition 1.7. Process Control Block (PCB)

The process state controls whether it is running, waiting or ready. The program counter locates the next instruction. The CPU registers contain the contents of the CPU registers i.e., CPU scheduling information (priorities, scheduling, queue pointers). Memory management information is the memory allocated to the process. Accounting information is the CPU used and time since start. I/O status is the list of open files I/O devices

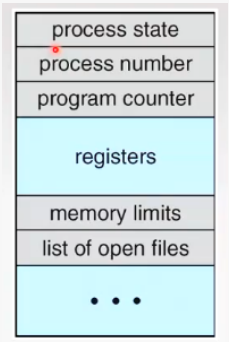


Figure 6: PCB of a process

The CPU can use the PCB to switch between processes as follows

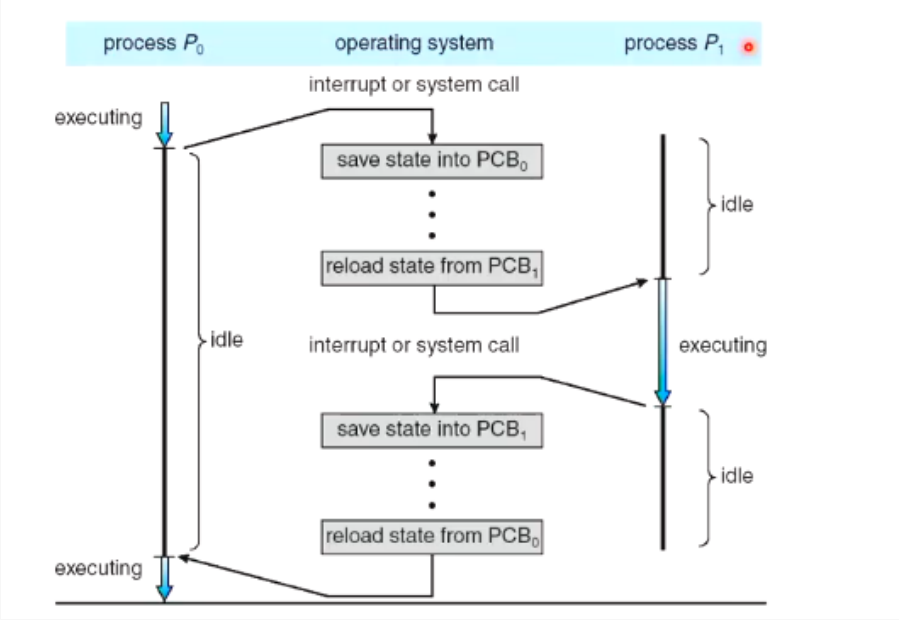


Figure 7: CPU switching between processes

During a switch, it is a overhead and we attempt to minimise the amount of switches as much as possible.

Definition 1.8. Process Scheduling

To maximise CPU utilisation, most modern OS support multi-programming, i.e., the ability to run multiple programs concurrently. Process scheduler selects among available processes for next execution on CPU. Some examples include:

- Job queue - set of all processes in the new state
- Ready queue - set of all processes in the ready state
- Device queue - set of processes waiting for an I/O device

Definition 1.9. Short-term Scheduler

The short-term schedule is invoked frequently, at least once in every 100ms. The short term scheduler must be fast. If it takes 10ms to decided to process a burst for 100 ms, then 9% $(\frac{10}{110})$ of the CPU is wasted

Definition 1.10. Long-Term Scheduler

The long-term schedule is much less frequent - may be minutes between creating one process and next. The long-term scheduler controls the degree of multiprogramming (number of processes in memory). Processes can be described as either I/O bound (spends more time doing I/O then computation) or CPU bound (Long CPU bursts). A good long-term scheduler would have a good process mix. Time-sharing systems like Linux and Windows don't have a long-term schedule. Everything is dumped to the short-term scheduler.

Definition 1.11. Process Creation

An OS must provide ways for users to create new processes. A system process (parent) creates a user process (child). The OS provides system calls for a user process to create another process. For example, in UNIX-based operating systems, processes can be created using the $fork\,()$ system call.

Example 1.2. Fork()

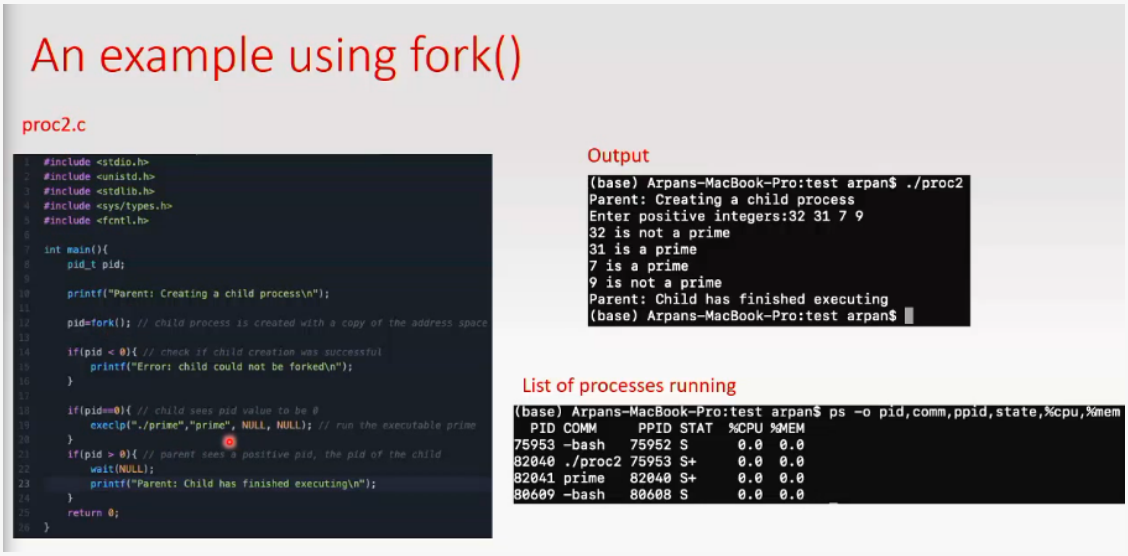


Figure 8: Fork example

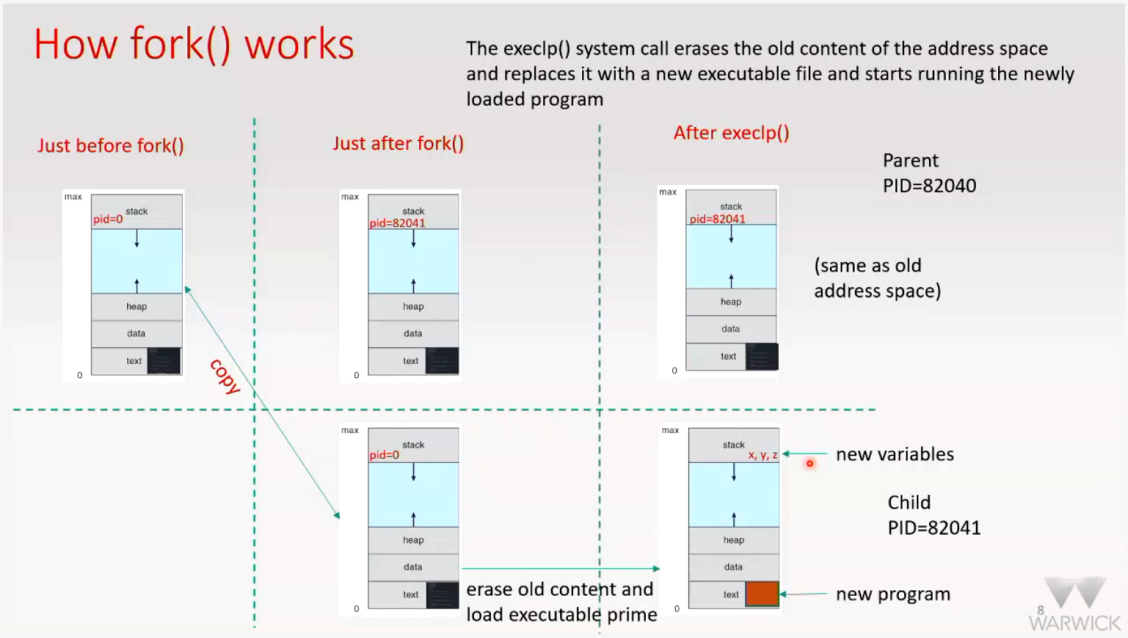


Figure 9: Fork system and execlp

Definition 1.12. Process Termination

A parent process may terminate the execution of child processes using the $abort\,()$ system call. Some examples include that child has exceeded its allocated resources. Task assigned to child is no longer required. The parent is exiting and the operating system does not allow a child to continue if its parent terminates.

1.3 Interprocess Communication (IPC)

Definition 1.13. Interprocess Communication (IPC)

Processes running concurrently may want to communicate with each other. For example, when two processes are working on a task together, they may need to share data. The data produced by one process may be required by other processes. There are two ways to achieve this:

- Shared memory - a region of memory that is shared by communicating processes is established
- Message passing - messages are exchanged between the communicating processes

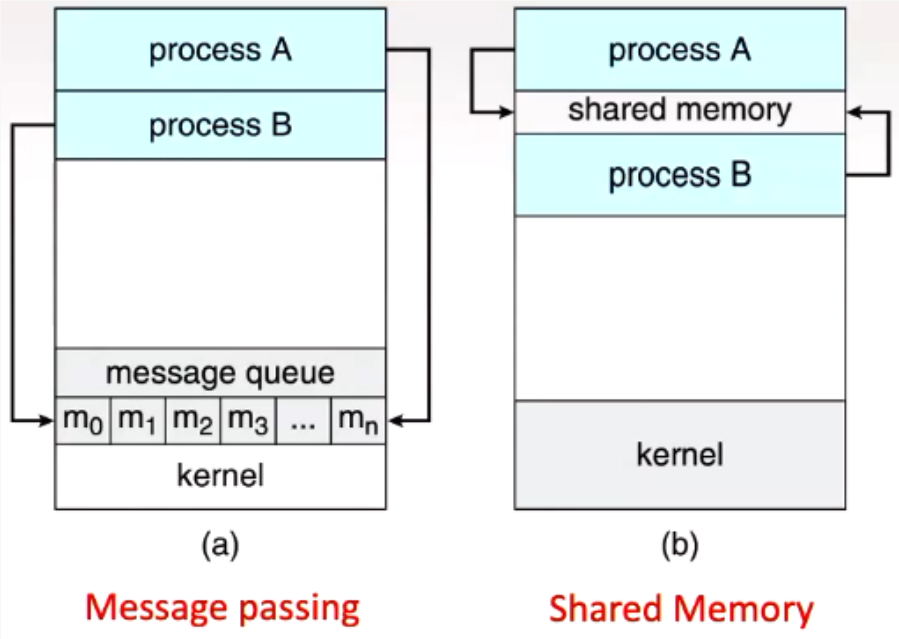


Figure 10: Message Passing vs Shared Memory

For message passing, processes interact by sending messages to each other. The kernel provides a logical communication channel and systems calls are used to pass messages.
For shared memory, processes interact by writing to or reading from a shared part of the memory. The kernel is involved only in establishing the shared part of memory; communication is entirely handled by the communication processes.

Definition 1.14. Shared Memory

Shared memory resides in the address space of one of the communicating processes. Other process need permission to access it. Kernel is required to setup shared memory and grant permission. Once shared memory is established, processes are responsible for maintaining proper synchronisation.

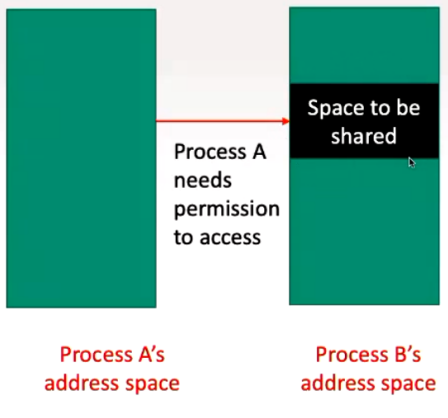


Figure 11: Shared Memory

Definition 1.15. Producer-Consumer Paradigm

The shared buffer/memory space can be filled by the producer and emptied by a consumer. Consumer must wait when the buffer is empty. Producer must wait when the buffer is full. When the buffer space is practically unbounded, this is not an issue.

Definition 1.16. Message Passing Systems

A message passing system should at least provide two operations:

- send(message)
- receive(message)

These are generally implemented using system calls. Processes must name each other explicitly: send (P, message) and receive(Q, message) or receive(id,message). We have a message in the receive so we can store the message. Links are established automatically and a link is associated with exactly one pair of communicating processes. Hard coding the processes identifier may not be ideal since every time a process is run its id can change. Or, it can be created using a mailbox system where send(mailboxid, message) and receive(mailboxid, message) is implemented. A shared mailbox is found in kernel with a specific ID named mailboxid. This is called indirect communication as we specify the mailbox instead of the process IDs. We can synchronise these messages by using blocking.

- Blocking send - the sender is blocked until the message is received
- Blocking receive - the receiver is blocked until a message is available

Definition 1.17. Buffering

Communication link is a buffer. Implementation of send() and receive() depends on the capacity of this buffer

- Zero capacity - the queues a maximum length of zero, so sender must block until recipient receives the message
- Bounded capacity - the queue has a finite length, when full the sender must be blocked
- Unbounded capacity - the queue's length is potentially infinite. Sender never blocks.

Example 1.3. Examples of IPC

Definition 1.18. mmap

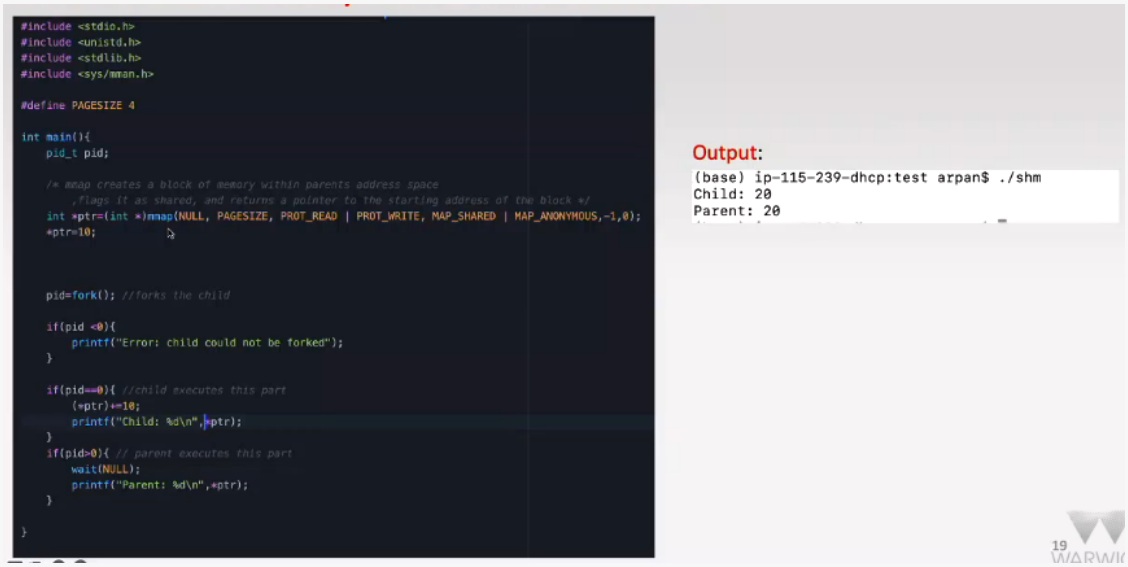


Figure 12: Shared Memory Communication using mmap

The mmap allocates memory in the memory address. Note that malloc uses mmap in its implementation.

Definition 1.19. Ordinary Pipes

Ordinary pipes in UNIX allow simple one-way communication through message passing. A pipe connects the output of one process to the input of another process. In UNIX systems, pipes are commonly created within bash terminal using |.

Figure 13: Ordinary pipe

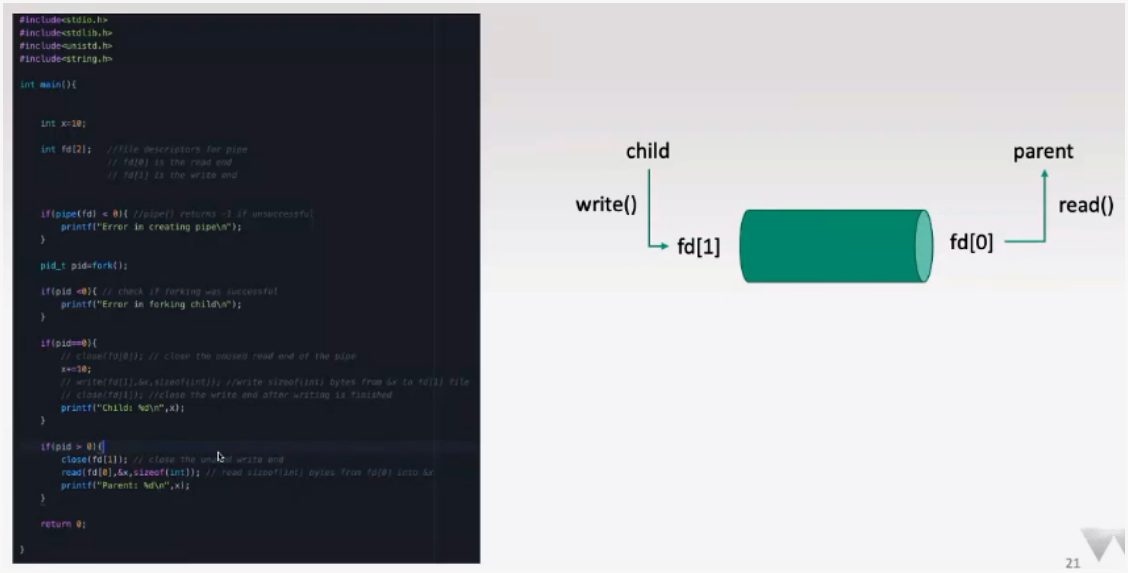


Figure 14: Ordinary Pipe in C

In the figure above, fd is a pipe that is used by the child created using fork. There is an error in the code where write should be uncommented. The write line inserts the new value of x into fd[1].

Definition 1.20. Named Pipes

```
(base) ip-115-239-dhcp:test arpan$ mkfifo namedpipe
(base) ip-115-239-dhcp:test arpan$ ls -la namedpipe
prw-r--r--  1 arpan  staff  0 13 Oct 16:23 namedpipe

(base) ip-115-239-dhcp:test arpan$ ./prime 32 31 19 9 8 > namedpipe
(base) ip-115-239-dhcp:test arpan$

(base) ip-115-239-dhcp:test arpan$ cat namedpipe
32 is not a prime
31 is a prime
19 is a prime
9 is not a prime
8 is not a prime

(base) ip-115-239-dhcp:test arpan$ rm namedpipe
(base) ip-115-239-dhcp:test arpan$
```

Figure 15: Named Pipes in UNIX

Named pipes are similar to files that store information, except they're not files and are separate allowing them to be faster.

1.4 Threading

Definition 1.21. Thread

A thread is a unit of CPU execution. Single-threaded process: one chain of execution running each line sequentially. If we have multiple similar tasks, one solution is to run the same function in a loop for each task. To achieve concurrency, create a separate process for each task, but it's not efficient. Each process requires its own address space in memory. Code, data could be shared. The benefits of threads include

- Economy - cheaper than process creation, thread switching lower overhead than context switching
- Scalability - large number of concurrent tasks
- Responsiveness - may allow continued execution of one thread even when another thread is blocked
- Resource sharing - threads share resources of process, easier than shared memory or message passing i.e., easy communication between threads

Definition 1.22. Multithreading

Multithreaded programs are when each thread perform a separate task from a single process. They share the same code, data and files as they're from a single process. Threads share more things with their parents than processes do i.e., code, data, heap, opened files, signals etc. Each thread is comprised of a thread id, program counter, a register set and a stack. A thread creation has less overhead as it shares a lot of things.

Definition 1.23. Concurrency vs Parallelism

Concurrency supports more than one task making progress. A single CPU system may appear to be running tasks concurrently by interleaving their execution.
Parallelism implies that a system can perform more than one task simultaneously.

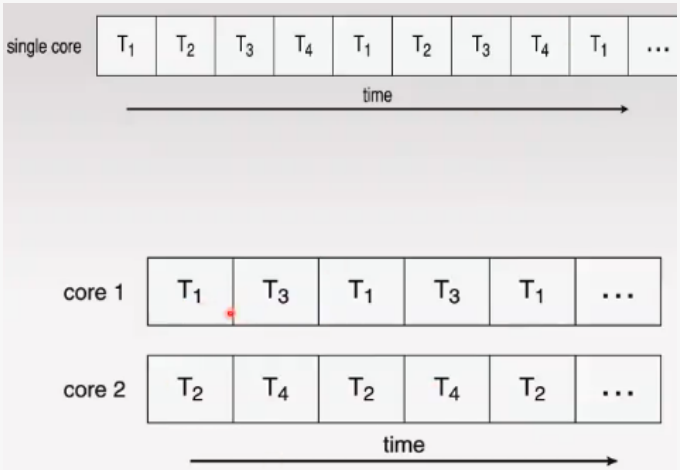


Figure 16: Concurrency vs parallelism

Definition 1.24. Data parallelism and task parallelism

Data parallelism distributes subsets of the same data across multiple cores, performing the same operation each core.
In task parallelism, it splits threads performing different tasks across multiple cores.
Applications can have a mix of both types.

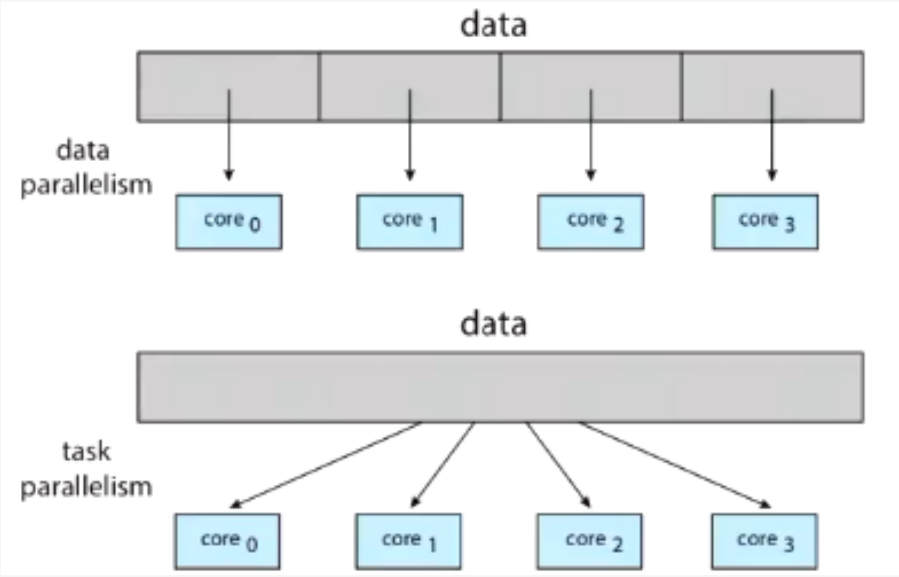


Figure 17: Data vs task parallelism

Theorem 1.1. *Amdahl's Law*

Amdahl's law helps us compute the speed up obtained by parallelism

$$\text{speedup} \leq \frac{1}{S + \frac{1-S}{N}} \tag{1}$$

Where time taken to run the serial part is $0 \leq S \leq 1$ and the 1 in numerator expresses the time taken before parallelising. The $\frac{1-S}{N}$ is the time taken to run the parallelisable part with N cores

Definition 1.25. Thread Synchronosation

When multiple threads write to the same location, we must synchronise the threads. Synchronisation ensure that one thread does not overwrite the contents written by the other thread.

Example 1.4. Thread Sync

Assume two threads, Thread 1 and Thread 2. They both try to update a global variable, sum. Thread 1 does

$$sum = sum + i$$

Thread 2 does

$$sum = sum + j$$

Assume they do this in a concurrent manner. Atomically, we have

$$\begin{aligned} register1 &= sum & register2 &= sum \\ register1 &= register1 + i & register2 &= register1 + j \\ sum &= register1 & sum &= register2 \end{aligned}$$

which results in wrong result, that is, for example, where $sum = 0$, $i = 1$ and $j = 5001$

Step	Thread	Action	Value
1	Thread 1	register1 = sum	register1 = 0
2	Thread 1	register1 = register1 +1	register1 = 1
3	Thread 2	register2 = sum	register2 = 0
4	Thread 2	register2 = register2 + 5001	register2 = 5001
5	Thread 2	sum = register2	sum = 5001
6	Thread 1	sum = register1	sum= 1

Figure 18: Order of operations without sync

From the table above, we can see that the value from Thread 2 is lost. This is error is called Race condition. Threads engage in a race to become the last one to write on the shared variable sum. Only one of the values is preserved. Race conditions should be avoided by proper synchronisation.

Definition 1.26. Mutex Locks

There are many ways to synchronise threads. One common way is use mutual exclusion locks or mutex locks. The idea is that each thread must first acquire a lock to perform updates on shared variables, called the critical section.

```
do {
    acquire lock
        update shared variables
    release lock
        remainder section
} while (TRUE);
```

Figure 19: Mutex Lock

Definition 1.27. User-level and Kernel level threads

Kernel itself may be multi-threaded and some kernel threads provide services to users and others are used to run user processes. Kernel can schedule them on different CPUs.

For user-level threads, they exist within a user process if they are multi-threaded. For a user-level thread to execute on a CPU, it must be associated with a kernel-level thread.

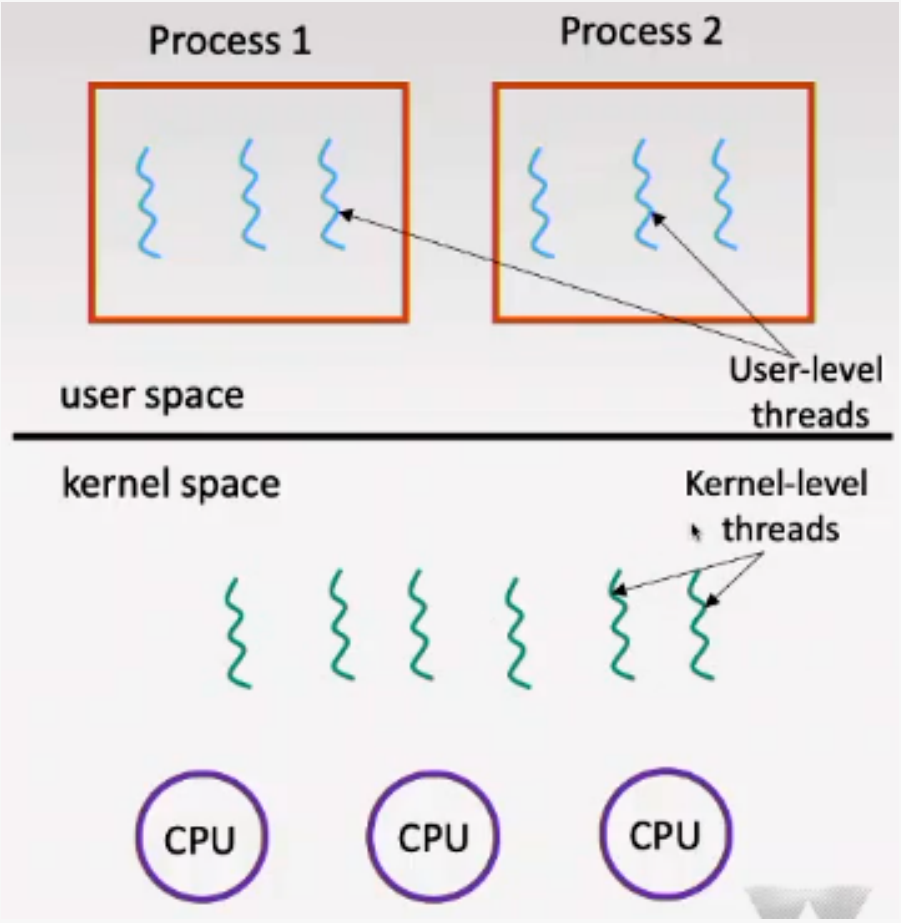


Figure 20: User-level and Kernel-level threads

Definition 1.28. One-to-one model

Each user-level thread maps to a kernel thread. Linux and windows use this. The advantages include

- Kernel is aware of all the threads running within the user process. Hence, we can rely on the kernel to schedule these threads on different CPUs.

The disadvantage includes

- Creating each user-level thread involves the kernel, meaning it is expensive.
- User is limited by the support provided by the kernel to manage threads.

Definition 1.29. Many-to-one model

The many to one model is when many user level threads are mapped to a single kernel thread. Advantages are:

- Less overhead and more flexibility in thread management

The disadvantages include:

- Multiple threads may not run in parallel, because only one may be in kernel at a time
- One blocking thread causes all to block

Definition 1.30. Many-tomany model

Mixture of one-to-one and many-to-one models. Allow user-level threads to be multiplexed onto smaller or equal number of kernels level threads. The advantages include

- Kernel threads can run in parallel, blocking call by one user-level thread does not block the entire process
- Programmers can decide how many kernel threads to use and how many user level threads should be mapped to each one, causing less overhead.

Definition 1.31. One thread-per-request strategy

Server creating a separate thread to handle each client request. This causes

- High overhead for each thread
- High traffic will create a large number of threads, slowing down the system by burdening it.

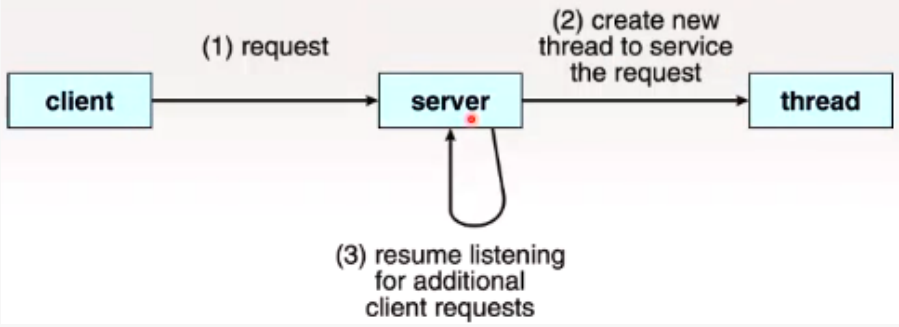


Figure 21: 1 TPS strategy

Definition 1.32. Thread Pool Strategy

The thread pool strategy involves the main server thread, which creates a predefined number n of worker threads. The number n usually does not change. Instead, we map existing n threads to serve new requests. When pool of thread workers are full, the main server thread creates a queue of pending requests.

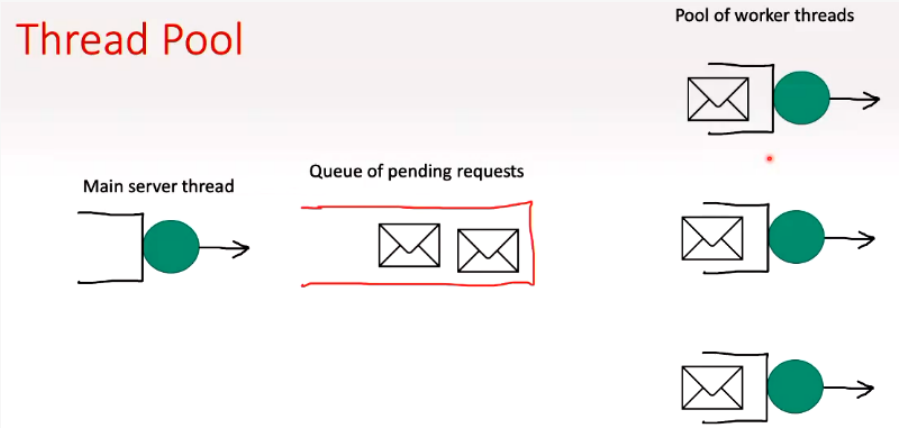


Figure 22: Thread Pool

The advantages include

- Usually slightly faster to service a request with an existing thread than create a new thread
- Allows the number of threads in the application to be bound to the size of the pool. This size is decided by the programmer taking into account the number of processing cores, memory in use, expected number of client requests etc.

Disadvantages are

- A request may have to wait in a queue if all workers are busy
- Deciding the number of workers in a threadpool is not easy. May have to be adjusted dynamically depending on load.

Definition 1.33. Signal Handling

Signals are used in UNIX systems to notify a process about the occurrence of certain events. Type of signals include synchronous i.e., internally generated by a process such as division by 0. And asynchronous which is externally generated such as terminating a process with CTRL+C sends signal SIGINT. All signals follow the same pattern: it is generated, delivered to the process, and handled by the process. Signal is handled by one of two signal handlers:

- Default - every signal has default handler that kernel runes when handling signal e.g. SIGINT
- User-defined - user defined signal handler can override default

In multi-threaded programs, a signal be delivered to all threads or specific threads. Synchronous signals are generally delivered to thread generating the signal. Some asynchronous signals like SIGINT should be sent to all threads. Signal can also be delivered to a specific process.

1.5 Scheduling

Definition 1.34. CPU and I/O bursts

Goal of CPU scheduling is to increase CPU utilisation. Life of a process is CPU burst followed by I/O burst. When a process is in a I/O burst, the CPU can be scheduled to some other process.

Definition 1.35. Performance Measures

The performance measures for CPU include:

- CPU Utilisation - fraction/percentage of time CPU remains busy when there are jobs in the ready queue
- Throughput - number of processes that complete their execution per time unit
- Turnaround time - amount of time to complete a process
- Waiting time - amount of time a process spends waiting in the ready queue
- Response time - amount of time it takes from when a request was submitted until the first response is produced

Definition 1.36. Types of scheduling

There are two types of scheduling:

- Non-preemptive - once the cpu is given to a process, the process holds onto the CPU until its current CPU burst finishes.
- Preemptive - the execution of a process is interrupted in the middle to schedule another process

Most modern OS use preemptive algorithms. Preemptive algorithms, however, have their own issues. Preemptive scheduling can cause race conditions. If a process is pre-empted while it is updating some shared data then it leaves data in an inconsistent state. Another process is scheduled which accesses the same data then it will leave it in an inconsistent state again. Shared data should be updated within critical sections using proper synchronisation primitives.

We will now discuss scheduling algorithms

Definition 1.37. First Come First Serve (FCFS)

Processes are assigned to the CPU in order of their arrivals.

Process	Burst Time (ms)
P1	24
P2	3
P3	3

Figure 23: FCFS Scheduler

The waiting time for the above information is 0 ms for P1, 24 ms for P2 and 27 ms for P3, meaning that the average waiting time is

$$\frac{0+24+27}{3} = 17\text{ms}$$

Definition 1.38. Shorest Job First (SJF)

The process with the shortest next CPU burst is selected. (If there is a tie, we use FCFS).

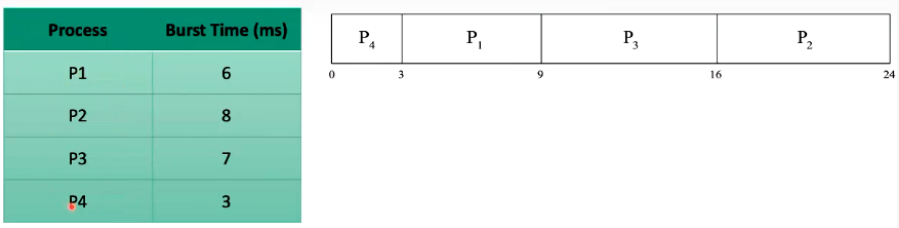


Figure 24: SJF

The average waiting time is

$$\frac{0+3+9+16}{4} = 7\text{ms}$$

In comparison, FCFS would've computed us an average of 10.25 ms. SJF is provably optimal, in that it gives the minimum average time. The problem is knowing how long the next CPU burst will be can only be an estimate. There are also two versions of SJF, which are pre-emptive and non-pre-emptive. Pre-emptive SJF switches to newly arrived process and non-pre-emptive SJF will allow the current existing job to finish.

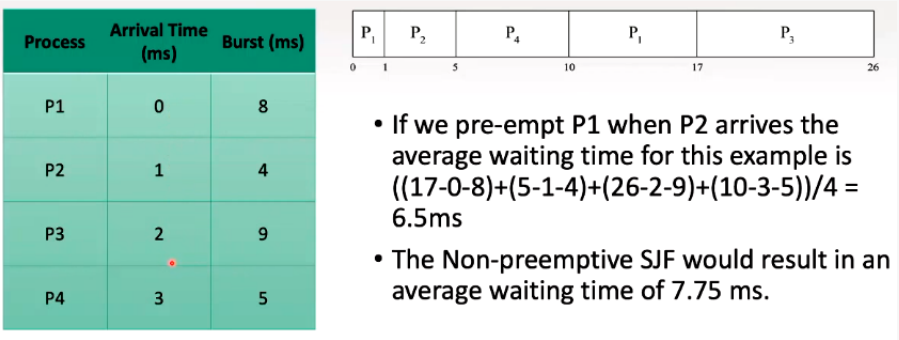


Figure 25: Preemptive SJF

Definition 1.39. Priority Scheduling

SJF is a special case of priority scheduling. A priority is associated with each process and CPU is allocated to the highest priority process. Priorities can be indicated by numbers. In SJF, priorities are the next CPU burst times.

However, priority schedule comes with a problem called starvation. Very low priority processes may never get scheduled. A fix to this is aging, gradually increasing the priority of processes that wait in the system for a long time.

Definition 1.40. Round Roubin (RR) Scheduling

Each process gets a small unit of CPU time called time quantum q. After this time has elapsed, the process is preempted and added to the end of the ready queue. Generally, the schedule visits the processes in order of their arrivals. If there are N processes in the ready queue and the time quantum is q, then each process gets $\frac{1}{N}$ of the CPU time in chunks of at most q time units at once. No process waits more than $(N-1)\,q$ time units for its next turn. Note that RR is naturally preemptive. If q is large, then it imitates the behaviour of FCFS. If q is small, then there are too many context switches. q is usually 10 ms to 100 ms.

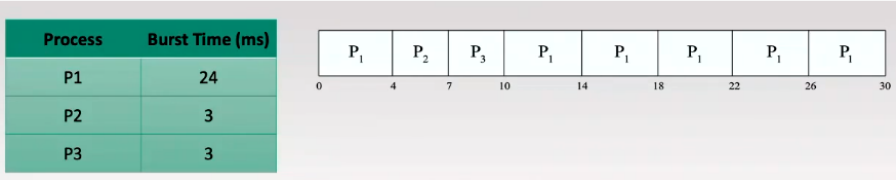


Figure 26: RR schedule with $q = 4$

In the above figure example, we have that the average waiting time is

$$\frac{(30-0-24)+(7-0-3)+(10-0-3)}{3} = 5.66\text{ms}$$

The waiting time using FCFS is 17 ms but SJF is 3 ms. Despite there is higher average waiting time than SJF, there is better response.

Theorem 1.2. *Exponential Moving Average*

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value of the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

Expanding the recursion gives

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \ldots + (1-\alpha)^j \alpha t_{n-j} + \ldots + (1-\alpha)^{n+1}\tau_0 \tag{2}$$

1.6 Synchronisation

Synchronisation is required for threads as discussed before. However, processes may need it as well e.g., shared memory. In this case, we will refer to processes as both processes and threads, as the solutions and problems are the same.

Definition 1.41. Critical Section

Critical section is part of the code where shared variables are updated.

When one process is in a critical section, no other process should be allowed in its critical section (mutual exclusion).

Definition 1.42. The Critical Section Problem

Consider a group of process

$$\{P_0, P_1, \ldots, P_N\}$$

Each process has a critical section of code where they update some shared variables. The critical section problem:
Design a protocol such that no two processes can concurrently execute their critical sections.
Mutual exclusion is not the only criteria that we must satisfy. We must also catch its progress. If no process is executing in its critical and there exist some processes that wish to enter their critical section, then one of the waiting processes must be able to enter into its critical section. The last criteria is the bounded waiting criteria. No one process should have to wait indefinitely to enter its critical section while other processes are being allowed to enter and exit their critical sections continually. Note that the bounded waiting is a stronger requirement than progress requirement.

Example 1.5. Possible solution for two processes

Consider two processes P_0 and P_1. One basic solution is to create a shared variable called "turn", a 0 or 1.

```
P0: do {                          P1: do {
        while (turn !=0)                  while (turn !=1)
              ; // busy wait                    ; // busy wait
          //critical section                  //critical section
        turn = 1;                         turn = 0;
          //remainder section                 //remainder section
    } while (true);                   } while (true);
```

Figure 27: Turn solution

Whilst this solution provides mutual exclusion, if P_1 finished whilst P_0 still has to edit the critical section, P_0 will not be able to access its critical section. Another solution is proposed by Gary L. Peterson in 1981. The solution involves the following logic:

```
/* shared variables */
int turn;                          /* whose turn */
boolean flag[2]={false,false};     /* stores wish to enter CS */

P0:                                P1:
do {                               do {
    flag[0] = true;  /* express wish */      flag[1] = true;
    turn = 1;  /* allow other process */     turn = 0;
    while (flag[1] && turn = = 1)            while (flag[0] && turn = = 0)
          ;                                        ;
    //critical section                       //critical section
    flag[0] = false;  /* done*/             flag[1] = false;
    //remainder section                      //remainder section
} while (true);                    } while (true);
```

Figure 28: Peterson's Algorithm

Peterson's algo satisfies all three criteria but it is not perfect. It employs busy and may fail in modern architectures, as the order of array and turn may switch causing it to fail.

Definition 1.43. Locks

We now explore more practical techniques to solve the CS problem. All solutions are based on idea of locking, i.e., two processes can not have a lock simultaneously.

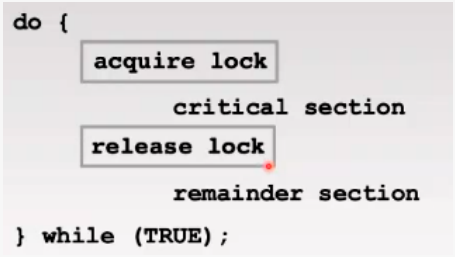


Figure 29: Locks

Locks should be atomic, meaning non-interruptible. Modern machines provide special atomic hardware instructions to implement locks. An example is test_and_set

Example 1.6. Test and set Instruction

The set and set instruction, by definition, returns original value of passed parameter $target$. Set the new value of passed parameter $target$ to "TRUE". It needs to be implemented atomically using hardware.

```
    /* shared variable */
    boolean lock=false;

do {
      while (test_and_set(&lock))
           ; /* do nothing */

          /* critical section */

          lock = false;

         /* remainder section */

} while (true);
```

Figure 30: Test and set solution

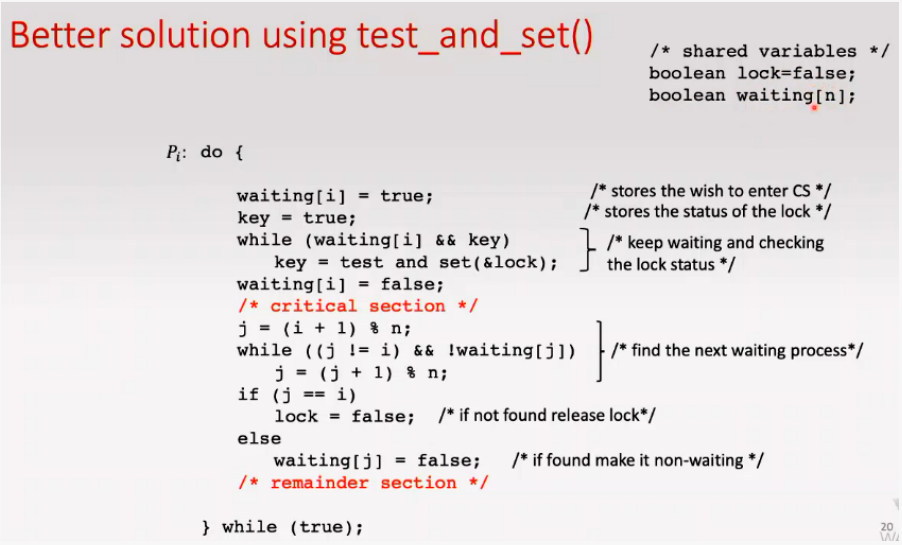


Figure 31: Better solution

Definition 1.44. Synchronisation Primitives

Hardware based solutions are generally inaccessible to programmers. OS designers build software tools to solve critical section problem. These are the synchronisation primitives:

- Mutex Locks
- Condition Variables
- Semaphores

Definition 1.45. Mutex Lock

```
do {
        lock (&mutex);          /* makes the caller wait for lock to be available,
                                   once available, grabs it and makes it unavailable */

        /* Critical Section */

        unlock(&mutex);         /* releases the lock */

}while(true)
```

Figure 32: Mutex Lock

The mutex lock has two operations available: the lock and unlock. The lock operation it makes other processes wait for the mutex lock to be available. Once it is available, a process is available to acquire that lock and makes mutex unavailable to other processes. It then executes the critical section and finally releases the lock. Functionally, the lock must be implemented with two fields: a boolean lock, and a list that list the processes waiting on the mutex.

```
struct mutex_lock{
     boolean lock;       // TRUE if available, FALSE otherwise
     struct list *list; // List of processes waiting on the mutex
}

lock(struct mutex_lock *mutex) {
   if (!mutex->lock){
      add process to mutex->list
      block();     // blocks the calling process
   }
   mutex->lock=FALSE;
}

unlock(struct mutex_lock *mutex) {

    remove a process P from mutex->list
    wakeup(P)
    mutex->lock=TRUE;
}
```

All operations must be performed atomically

All operations must be performed atomically

Figure 33: Mutex Lock Implementation

Initially, the lock command first checks if mutex is locked. If locked, it adds the process onto the list and we block that process, meaning that the process goes to sleep. The unlock operation removes a process P from the list, wakes up that process P, and sets the value of lock to true.

Definition 1.46. Semaphores

Semaphores can have integer values, meaning that they're more powerful than mutex locks. A zero value indicates that the semaphore is not available. Positive values indicate that it is available. It can only be accessed via two indivisible atomic operations $wait\,()$ and $signal\,()$. The operations are as following:

- $wait\,()$- checks if the value of the semaphore is ≤ 0 and if so it makes the calling process wait until it becomes positive
- Once the semaphore value is positive $wait\,()$ function decrements it by 1.
- $signal\,()$ function increments the value of the semaphore by 1
- Both $wait\,()$ and $signal\,()$ must be performed atomically

```
struct semaphore S;   // declaration
S.value=1;     // initialisation

wait(&S) // decrement to 0
/* Critical Section */
signal(&S) // increment to 1
```

Semaphore acts like a mutex lock

Figure 34: Semaphore as mutex

```
struct semaphore S;   // declaration
S.value=2;     // initialisation

wait(&S);
/* Access resource, e.g., buffer */
signal(&S);
```

Semaphore controls th number of processes that can concurrently access a resource

Figure 35: Semaphore with higher number

Semaphores can also be initialised with a higher number if we want processes to access shared memory simultaneously. The implementation is similar to that of mutex, however, we change the boolean statement to an integer instead as follows

```
struct semaphore {
     int value;       // semaphore value
     struct list *list; // List of processes waiting on the mutex
}

wait(struct semaphore *S) {
   if (S->value <= 0){
      add process to S->list
      block();     // blocks the calling process
    }
    S->value--;
}

signal(struct semaphore *S) {

    remove a process P from S->list
    wakeup(P)
    S->value++;
}
```

All operations must be performed atomically

All operations must be performed atomically

Figure 36: Semaphore implementation

Definition 1.47. Deadlock

Deadlock is when two or more processes waiting indefinitely for an event can only be caused by the waiting process. For example, A waits for B to do something. B waits for A to do something, therefore A and B are in deadlock. Let S and Q be two semaphores initialised to 1.

Definition 1.48. Starvation

Starvation occurs when a specific process has to wait indefinitely whilst others make progress. Starvation occurs when multiple processes are waiting on a semaphore and the signal call wakes up the same process again and again. To avoid such starvation signal should randomly pick the process to wake up.

Definition 1.49. Priority Inversion

Priority inversion is a scheduling problem when lower-priority process holds a lock needed by higher-priority process. It is solved via priority-inheritance protocol. Priority inheritance says that all processes that share the lock will share the same priority level.

Definition 1.50. Bounded Buffer Problem

This problem involves a buffer capable of storing n items. Producer producers items and writes them to the buffer. The consumer consumers items from the buffer. We want to sync the producer and the consumer. There are three requirements

1. Producer should not write when buffer is full
2. Consumer should not read when buffer is empty
3. Producer and consumer should not access the buffer at the same time

Therefore, our code would be something like

```
/* shared semaphores */
semaphore mutex=1;   /* maintains exclusive access to the buffer */
semaphore full=0;    /* counts the number of items in the buffer */
semaphore empty=n;   /* counts the number of empty slots in the buffer */

Producer: do {
            ...
           /* produce an item in next_produced */
            ...
          wait(empty);
          wait(mutex);
             ...
           /* add next produced to the buffer */
             ...
          signal(mutex);
          signal(full);
        } while (true);
```

Figure 37: Producer

```
Consumer: do {
            wait(full);
            wait(mutex);
               ...
            /* remove an item from buffer to next_consumed */
               ...
            signal(mutex);
            signal(empty);
               ...
            /* consume the item in next consumed */
               ...
          } while (true);
```

Figure 38: Consumer

Definition 1.51. Readers and Writers problem

A data set is shared among a number of concurrent processes. Readers only read the data set; they do not perform updates. Writers can both read and write. We want to allow multiple readers to read at the same time. However, only one single writer can access the shared data at the same time. There are many different versions involving different priorities. For our version, our readers are given preference over writers when no process is active. Writers may also starve. For the writer process we have

```
/* shared variables */
semaphore rw_mutex=1;    /* lock to allow at most one writer */
semaphore mutex=1;       /* lock to protect the read_count */
int read_count=0;        /* counts the number of reader processes */


writer: do {
          wait(rw_mutex);
             ...
          /* writing is performed */
             ...
          signal(rw_mutex);

     } while (true);
```

Figure 39: Writer Process

And for the reader, we have

```
/* shared variables */
semaphore rw_mutex=1;    /* lock to allow at most one writer */
semaphore mutex=1;       /* lock to protect the read_count */
int read_count=0;        /* counts the number of reader processes */

reader: do {
          wait(mutex);
          read_count++;
          if (read_count == 1)
            wait(rw_mutex);
          signal(mutex);
             ...
           /* reading is performed */
             ...
          wait(mutex);
          read_count--;
          if (read_count == 0)
           signal(rw_mutex);
          signal(mutex);
        } while (true);
```

Figure 40: Reader Process

Definition 1.52. Dining Philosophers

Philosophers spend their lives alternating thinking and eating. Don't interact wtih their neighbours, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl. They need both to eat, and release both when done. In the case of 5 philosophers, the shared data is the bowl of rice, the semaphore is the chopstick[5] all initialised to 1 and two neighbouring philosophers cannot eat at the same time.

```
Philosopher i :
do {
   wait (chopstick[i] );
   wait (chopstick[ (i + 1) % 5] );

              //  eat

   signal (chopstick[i] );
   signal (chopstick[ (i + 1) % 5] );

                 //  think

} while (TRUE);
```

Figure 41: Dining Philosophers solution

However, the solution above has a problem, where if all philosophers get hungry and they grab the left chopstick, they will be in a deadlock. There are however solutions to this. One way to fix is it is not to fill the table. For example, we allow at most 4 philosophers be sitting simultaneously at the table. We can also allow a philosopher to pick up the chopsticks only if both are available. Lastly, we can also use an asymmetric solution, use an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up the right chopstick and then the left chopstick

1.7 Deadlocks

Definition 1.53. Deadlock

A set of processes is in a deadlock when each process in the set is waiting for an event that can be caused only by another process in the set. The event never occurs. The events we are interested in are acquisition or release of some type of resources, for example mutex locks, semaphores, cpu, file, i/o devices etc.

Example 1.7. A quick example

Given two processes P_1 and P_2 and two locks L_1 and L_2, we first have L_1 being acquired by P_1. Then, with context switch, we have L_2 is acquired by P_2. Then, through another context switch, we run P_1 again only to find that it is waiting for L_2. Similarly, with another context switch to P_2, we find that P_2 is waiting for L_1. If events occur in a different sequence, we may not see a deadlock. Therefore we have developed systematic methods to detect deadlocks.

Definition 1.54. System Model

System consists of different types of resources $R_1, R_2, \ldots, R_m$ which are things such as mutex locks, CPUs, I/O devices etc. Each resource type R_i has W_i instances i.e., 3 mutex locks, 4 printers etc. A set of processes $\{P_0, P_1, \ldots, P_n\}$. Each process utilises a resource as follows: request, use, release.

Definition 1.55. Necessary conditions for Deadlock

1. Mutual Exclusion - only one process can use an instance of resource at a time
2. Hold and Wait - there must be a process holding some resources while waiting to acquire additional resources held by other process
3. No preemption - a resource can be released only voluntarily by the process holding it, after the process has completed its task
4. Circular wait - there must exist $\{\overline{P}_0, \ldots, \overline{P}_m\} \subseteq \{P_0, \ldots, P_n\}$ such that $\overline{P}_0$ is waiting for $\overline{P}_1$, $\overline{P}_1$ is waiting for $\overline{P_2}, \ldots, \overline{P}_m$ is waiting for $\overline{P}_0$

Definition 1.56. Resource Allocation Graph

A directed graph $G = (V, E)$ where V is partitioned into two types:

$$P = \{P_1, P_2, \ldots, P_m\} \text{the set consisting of all the processes in the system}$$
$$R = \{R_1, R_2, \ldots, R_m\} \text{the set consisting of all resource types in the system}$$

The request edge is defined as the directed edge $P_i \to R_j$
The assignment edge is defined as the directed edge $R_j \to P_i$.
We will illustrate our graph using process nodes as circles and resource type nodes as squares with black dots in them which denote instances. For example, a square with 4 black dots would denote a resource type with 4 instances.

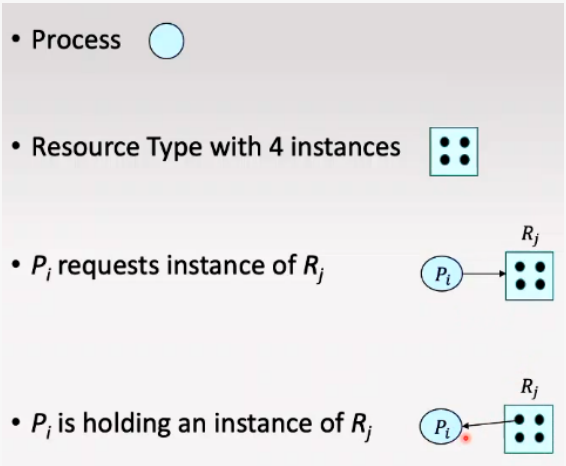


Figure 42: Illustration of our graph

Example 1.8. Resource Allocation Graph Example

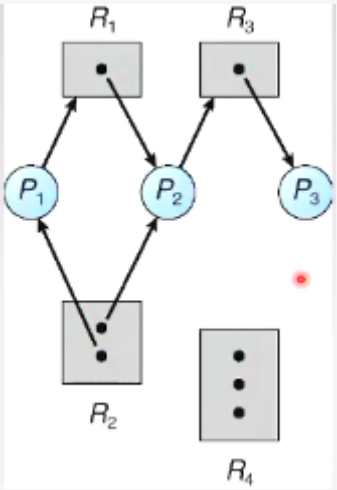


Figure 43: Example Graph

In the graph above, we have that P_1 has an instance of R_2 and has requested an instance from R_1. P_2 has instances of R_1 and R_2 but is waiting for instance of R_3. P_3 has instance of R_3. Is there a deadlock?
No. There are no cycles in the graph. Furthermore, even if there are smaller cycles in the graph, this does not indicated a deadlock. For a cycle to indicate a deadlock, none of the processes have all their requests satisfied.

Definition 1.57. Conclusion for Graph

If graph contains no cycle $\implies$ no deadlock
If graph contains a cycle $\implies$ need to look further. For small examples, we can manually detect deadlocks. But for larger examples, we need an algorithm

Definition 1.58. Deadlock Detecction Algorithm

The algorithm involves transferring the graphical information into a tabular form.

	Allocation			Currently Available			Request		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P1	0	1	0	0	0	0	1	0	0
P2	1	1	0				0	0	1
P3	0	0	1				0	0	0

Figure 44: Deadlock Detection Tabular Form

We further label that P_1, P_2 and P_3 are all unfinished, therefore their finish is false. However, we can set P_3 to true as it has no requests and reclaim the resource occupied for R_3. We can now satisfy P_2 request for R_3 and finish it as well. Lastly, we can finish P_1 now that we also have R_1 free.

Definition 1.59. Deadlock Prevention

We need to ensure that at least one of the necessary conditions for deadlocks does not hold. We cannot change the mutual exclusion rule, as it is required for some processes. Instead, we can modify the hold and wait rule, where we ensure a process either gets all or none of its required resources so it does not hold any other resources. We can also use no preemption, i.e., if a process holding some resources requests additional resources that cannot be immediately allocated to it, then all resources currently being held by the process can be released. Lastly, we can also modify circular wait. Number of resources and require that each process requests resources in an increasing order of enumeration. Process holding resource n cannot request a resource with a number less than n.

Definition 1.60. Deadlock Avoidance

Deadlock avoidance is less restrictive than deadlock prevention. It determines if a request should be granted based on if the resulting allocation leaves the system in a safe state. Safe state is where deadlock can never occur, no matter what future requests arrive. We can determine if a state is safe using the deadlock avoidance algorithm. It needs a priori advanced information on resource requirements. Each process declares the maximum number of instances of each resource type that it may need. Upon receiving a resource request, the deadlock avoidance algorithm checks if granting the resource immediately leaves the system in a safe state. If so, grant the request immediately, otherwise wait until the state of the system changes to a state where the request can be granted safely.

Definition 1.61. Banker's Safety Algorithm

Take 5 processes as an example denoted $P_0, \ldots, P_4$. There are 3 resource types, A, B and C which have 10,5 and 7 instances respectively. Consider the following table.

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Figure 45: Banker's Algorithm Table

Obtain additional need of each process, we do $max - allocation$. We can calculate $available = instances - \sum Allocation$. We now find processes whose additional needs satisfy currently available resources. For example, P_1 only needs 1 2 2 which means that we can execute it. We will execute it and reclaim all the resources occupied by P_1. Therefore, we now have available 5 3 2 resources. Next process we can satisfy is P_3 which now obtains us availability of 7 4 3, continually until we have no processes left to execute. The order of sequence we have found is called a **safe sequence**. If a safe sequence exists, it means that the starting state of the system is safe.

Definition 1.62. Resource Request Algorithm

With Banker's safety algorithm we can determine if a given state is safe. However, to determine if a request for resource can be granted immediately, we need to determine if by granting the request we are in a safe state. Therefore, we have the resource request algorithm. Pretend the request is granted and determine if the resulting state is safe using Banker's safety algorithm. If it is found to be safe, we grant the request, otherwise we keep the request pending until state change.

1.8 Memory

A program must be taken from disk into memory to be executed. Once in the memory, the process occupies some addresses which store instructions to run the process and data of input and output. The OS must allocate a unique set of addresses to each process. If not, then one process can modify data/instructions belonging to other processes. The OS must guarantee that a process is only able to access the addresses belonging to that process. During the execution of a process, the CPU needs to access different addresses belonging to the process.

Definition 1.63. Logical vs Physical Address Space

During the execution of a process, the CPU needs to access different addresses belonging to the process. Logical address generated by the CPU to fetch instructions or read/write data; may be different from actual physical address. The physical address is seen by the memory unit; a logical address must be converted to a physical address before a memory access. Lastly, MMU is special hardware, required to translate logical addresses to physical addresses.

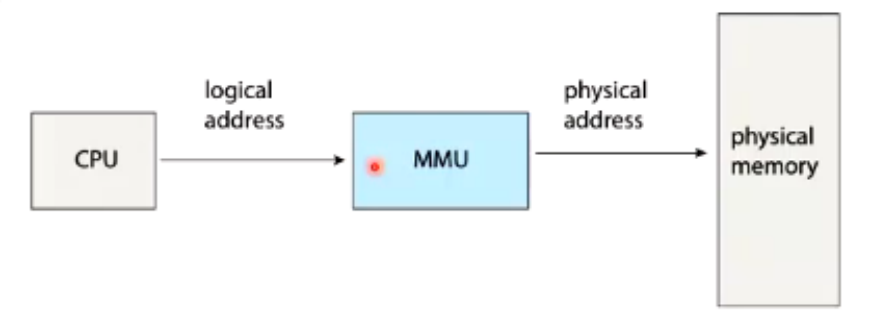


Figure 46: Logical to Physical

The reason we separate logical and physical address is to make programming easier. The programmer does not need to worry about how the process is actually laid out in the memory.

Definition 1.64. Memory Allocation

Memory allocation deals with how the OS allocates memory to processes and protects that memroy from other processes. We shall discuss three main techniques:

1. Contiguous memory allocation
2. Segmentation
3. Paging

Definition 1.65. Contiguous Memory Allocation

In contiguous memory allocation, each process is contained in a single section of memory starting from a base address. Each process occupies a contiguous block of addresses. This is a technique used in older OS's. To implement this scheme, the OS needs to decide the base address and the range of addresses for each process. The memory management unit (MMU) consists of relocation and limit registers storing the base address and range of addresses of a process. When a process is scheduled, the base and limit registers the loaded with the corresponding values by the OS.

Memory is divided into fixed size partitions. Each partitions is allocated to one process. When process terminates the partition is freed. There is a drawback, however, and that is the number of partitions in the memory fixes the number of concurrent processes in the memory.

In another case, where size partitions are variable. Any available block of memory is called a hole. Holes are scattered throughout memory. The OS keeps track of holes. When a process arrives, it is allocated memory from a hole large enough to accommodate it. Process exiting creates a hole, are combined into a single hole.

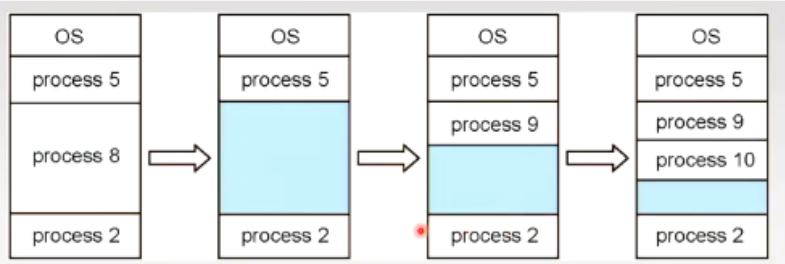


Figure 47: Partitions and holes

However, the variable comes with an inherent problem: we need to track and allocate holes to incoming processes. The solutions are as follows:

- First-fit - allocate the first hole that is big enough
- Best-fit - allocate the smallest hole that is big enough; must search entire list, unless ordered by size. It creates the smallest leftover hole
- Worst-fit - allocate the largest hole; must also search entire list. This produces the largest leftover hole.

Definition 1.66. Fragmentation

Memory allocation method can fragment the usable memory space. Consider external fragmentation where the total memory space exists to satisfy a request, but it is not contiguous. Variable partitioning method suffers from this problem. Keeping tack of a large number of small holes may have a high overhead.
Another problem might be internal fragmentation. Allocated partition may be much larger than requested memory; this size difference is memory internal to a partition, but not being used
In order to deal with the issue of fragmentation, we can use

- Compaction - shuffle memory contents to place all free memory together in one contiguous block
- Non-contiguous memory allocation - allowing a process to be scattered throughout the memory

Definition 1.67. Segmentation

For non-contiguous memory allocation, we divide a program into segments. Each segment is stored in a contiguous memory block. Each logical address consists of a two tuple

$$< segment - number, offset >$$

Segment number can be mapped to the base of the segment in the memory and offset is added to it to produce the complete physical address.
Segment table is indexed by segment numbers; each table entry has: segment-base and a segment-limit. The segment base contains where starting physical address where the segments reside in memory. The segment limit specifies the length of the segment

Definition 1.68. Paging

Segmentation cannot avoid external fragmentation. We can instead avoid it with paging. We divide program into fixed-sized blocks called pages. We then divide physical memory into fixed-sized blocks called frames. We now have

$$PageSize = FrameSize$$

We then assign pages to frames. The mapping between page numbers and frame numbers is stored in a page table. For each process the OS maintains a separate page table.

Definition 1.69. Logical Address under Paging

Address generated by CPU is divided into

- Page number (p) - used as an index into a page table which contains base address of each page in physical memroy
- Page offset (d) - combined with base address to define the physical memory address that is sent to the memory unit.

If logical address is m bits and pages size is 2^n bytes then last n bits is used to denote the page offset.

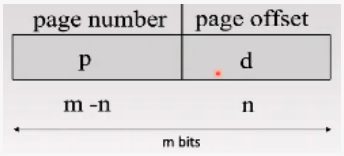


Figure 48: Table breakdown

In the figure below, observe how address is retrieved: the p is indexed in the page table to obtain the frame address f. It is then concatenated with the number from d, which is the logical address.

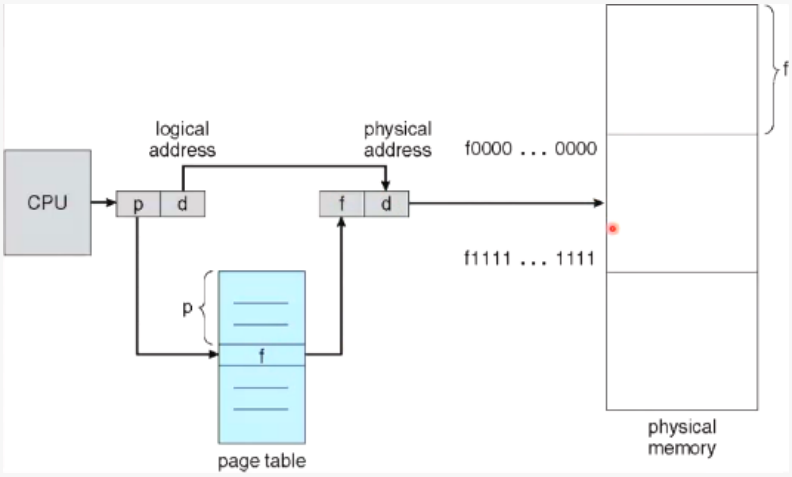


Figure 49: Paging Retrieval

Notice that internal fragmentation cannot occur as we divide frames and pages perfectly. Except, however, the last leftover frame with remainder bytes will be the only wasted bytes.

We will now discuss how page translation is implemented in an OS using hardware.

Definition 1.70. Directly from the memory

When a CPU generates page number and offset address for a process, we will use that page number to index into the page table. The page table itself is stored in the memory, therefore we need to locate the page table. We need to store the base address of a page table in the PCB. When a process is scheduled the base address of the page table is loaded into a register called page table base register (PTBR). Page number is added to the base address to find the location in memory where the corresponding frame number is stored.

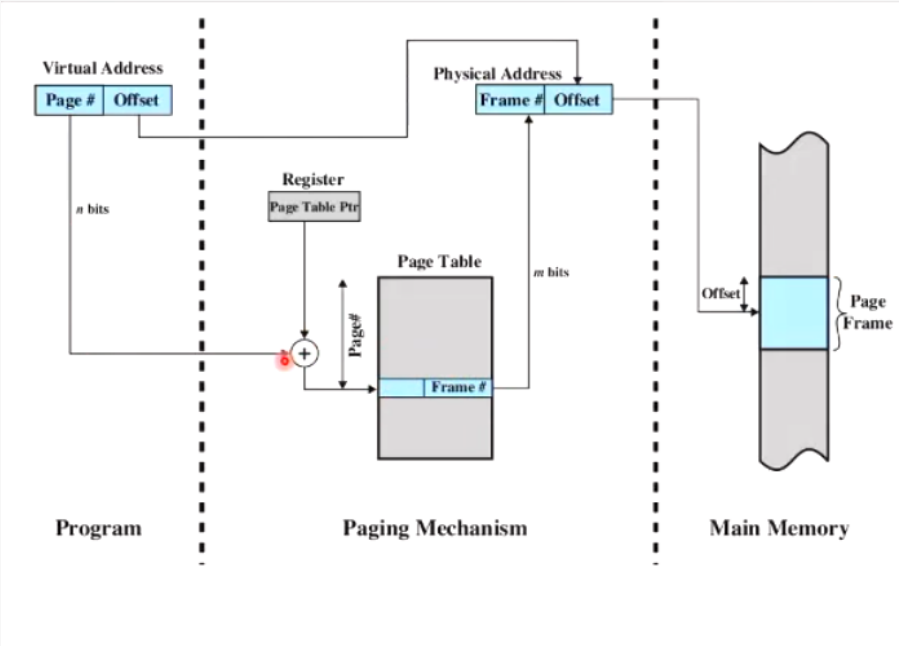


Figure 50: Illustration

The drawback is that at least two memory accesses are required to fetch every instruction/data, increasing overhead.

Definition 1.71. Translation Lookaside Buffer (TLB)

Translation lookaside buffer is a hardware cache to store frequently used page table entries. It stores a small number of entries, < 256. When the CPU generates a logical address its page number is checked against all the entries in TLB in parallel to find a match.

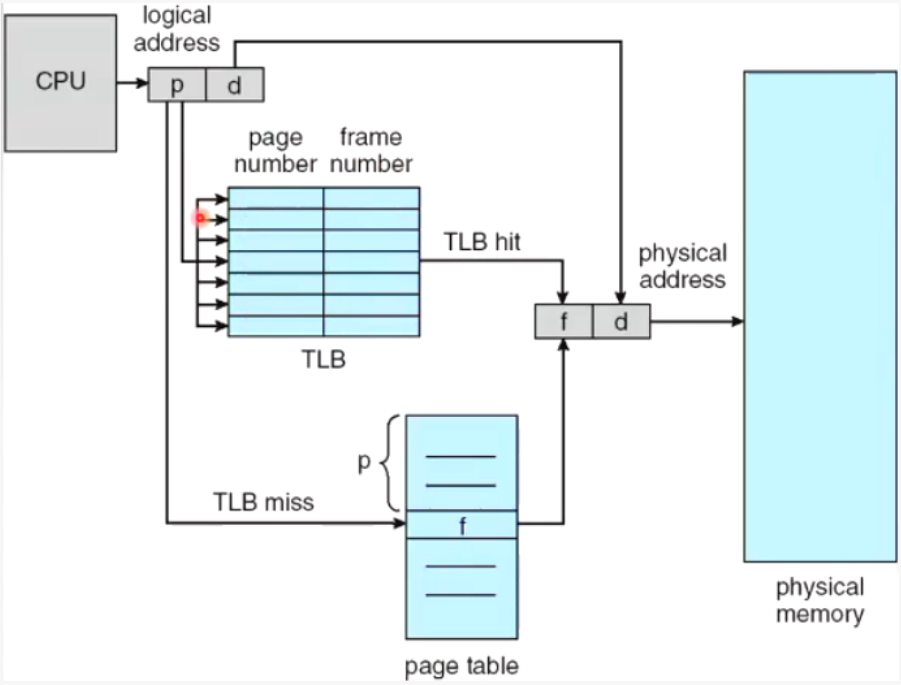


Figure 51: TLB

TLB is small and is cache memory, therefore the search is really fast. If TLB misses, then we try to find our page using the page table. We then fill the TLB with that specific entry we were looking for in case for the future. The TLB can store page table entries of multiple processes. However, each entry of the TLB requires an address space identifier (ASID) to uniquely identify the process requesting the TLB search. If ASIDs match, then the frame number is returned; otherwise the request is considered to be a cache miss, which also guarantees memory protection.

Definition 1.72. Effective Access Time

The effective access time is defined as the following

$$\text{Effective Access Time} = \text{Hit Ratio} \times 1 \times \text{Memory Access} + (1 - \text{Hit ratio}) \times 2 \times \text{Memory Access} \quad (3)$$

Example 1.9. Access Time Example

Given a 95% hit ratio and a 100 ns access time, what is the effective access time? We have

$$\begin{aligned} EAT &= 0.95 \times 100 + 0.05 \times 200 \\ &= 105\text{ns} \end{aligned}$$

Definition 1.73. Smaller Page Tables

It is beneficial to have small page tables, it helps us reduce the memory overhead and reduce the page search time in case of a TLB miss. Most modern systems use 32 or 64 bit logical addresses. With 32 bit addresses and a 4 KB page size, the number of entries is 2^{20}. If each entry requires 4 bytes, then the size of a page table is 4 MB. Imagine the memory overhead if we are to run 100 processes.

Definition 1.74. Hierarchal Multi-level paging tables

We divide the page table into pages and store each page in a frame in the memory. That is, we store different pages in pages, compressing our original paging table. However, we will have to store an outer page table to store our newly created pages that map to our original pages. If the outer page table is bigger than one frame, we can further divide it into pages. The OS does not need to store the inner PTs not in use. The flat page requires 4 MB of space which is larger than 4 KB that is the page size. To store the inner page tables we would need $2^{10} = \frac{4MB}{4KB}$ frames. The outer page table will therefore have 2^{10} entries of 4 bytes each. Total size is 4 KB. Fits into one frame. No further paging required. Then, addressing is know defined as follows:

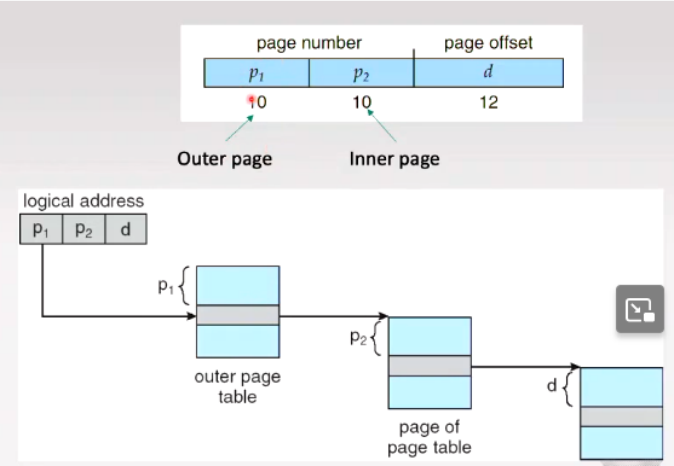


Figure 52: Addressing with Multi-Level Paging

However, now we have 3 memory accesses instead of 2. Multi-Level paging reduces the memory overhead but increases the number of memory accesses required to read/write data in case of a TLB miss.

Definition 1.75. Hashed Table Maps

Page numbers serve as hash keys and are hashed at an index of the PT. Each entry is a pointer to a linked list of page numbers having the same hash value. Each node contains the page number, the frame number and a pointer to the next node. Page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

2 Networks

2.1 Components of a network

Definition 2.1. Network

A network of inter-connected computing devices which enable processes running on different devices to communicate. Processes communicate by sending messages. Intermediate nodes help forward messages. For example, the internet.

Definition 2.2. Network Edge

Consists of end hosts which run network applications, e.g., Web, Email, Video Streaming, Online games.

Definition 2.3. Network core

consists of packet switches which help forward data packets e.g., switch and router

Definition 2.4. Communication links

carry data between network devices as electromagnetic waves. E.g., fiber, copper, radio, satellite links

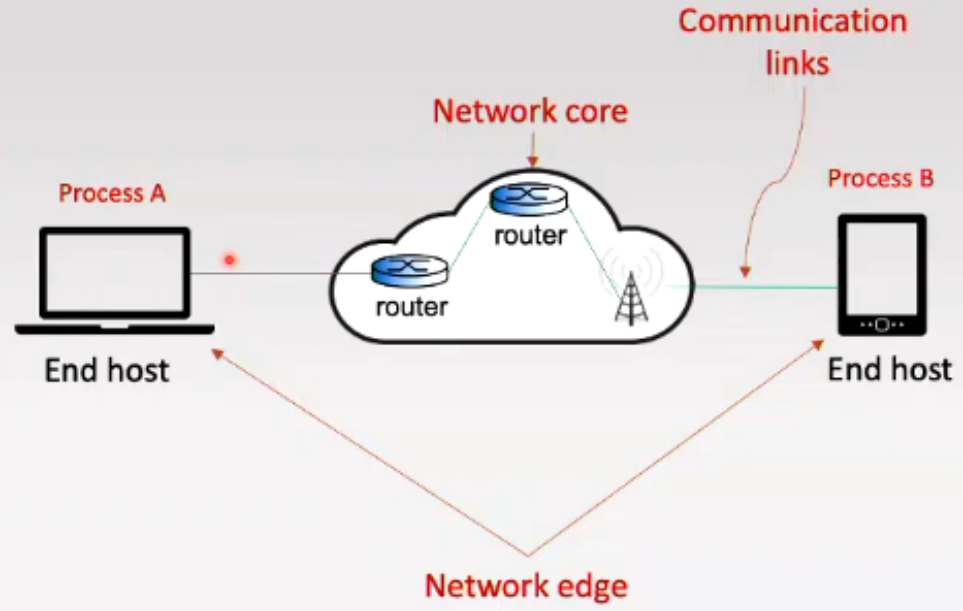


Figure 53: Components of a network

Definition 2.5. End-host

End-hosts run application processes e.g., mail, web which generate messages. It breaks down application messages into smaller chunks called packets. It adds additional information e.g., IP address, port number as packet headers so that the packets can be carried by the internet to their destinations. The IP address uniquely identifies an end host in the network and a port number uniquely identifies a process running within an end host. It then sends these bits over a physical medium. If needed provides reliable and orderly delivery of the packets. It also controls the rate of transmissions of packets

Definition 2.6. Network Core

The network core has a function of routing - run routing algorithms to construct routing tables. It also has a function of forwarding - once a packet arrives, it is forwarded to the appropriate output link according to the routing table.

2.2 Delay

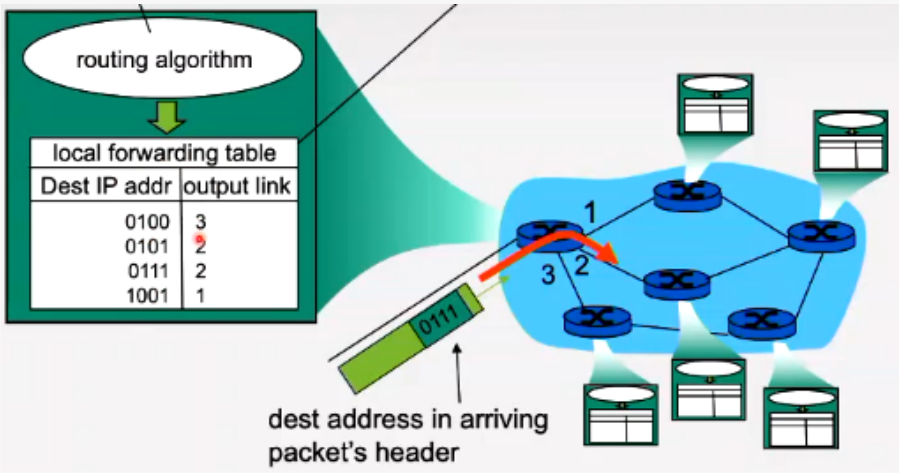


Figure 54: Routing

Definition 2.7. Store-and-forward principle

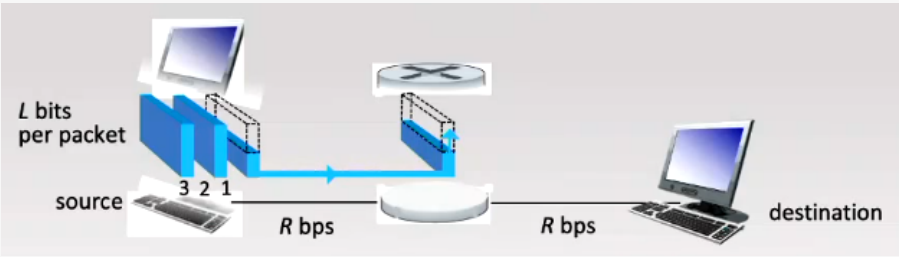


Figure 55: Store-and-forward principle

The store and forward principle consists of the idea that an entire packet must arrive at router before it can be transmitted on next link. It takes $\frac{L}{R}$ seconds to transmit L-bit packet into link at R bps. The end-to-end delay is $\frac{2L}{R}$.

Definition 2.8. Queueing delay and loss

If arrival rate (in bits) to link exceeds transmission rate of link for a period of time, packets will be in queue waiting to be transmitted on link or packets can be dropped (lost) if memory fills up.

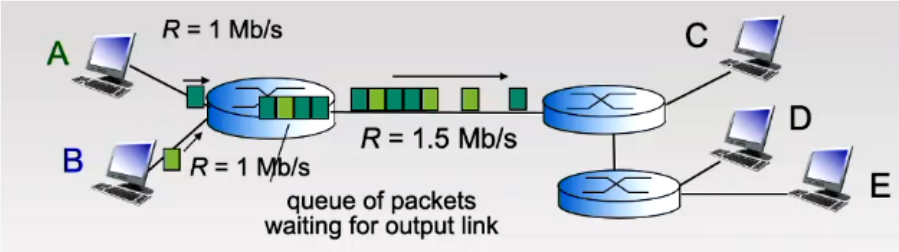


Figure 56: Queueing delay

Theorem 2.1. *Four sources of delay*

There are sources of delay:

1. d_{trans} - transition delay - $d_{trans} = \frac{L}{R}$ - where L is packet length (bits) and R is link bandwidth (bps)
2. d_{queue} - queueing delay - time waiting at output link for transmission (depends on congestion level of router)
3. d_{proc} - nodal processing - checks bit errors, determines output link and typically $< msec$
4. d_{prop} - propagation delay - $d_{prop} = \frac{d}{s}$ - where d is length of physical link and s is propagation speed in medium (approx $2 \times 10^8 \frac{m}{s}$)

The final formula is

$$d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop} \tag{4}$$

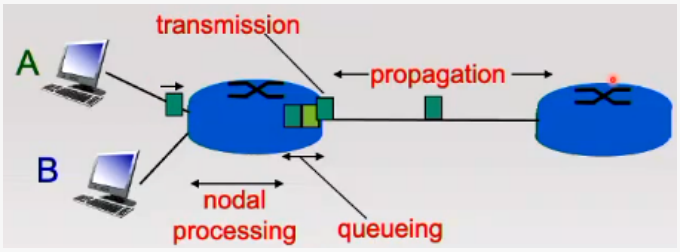


Figure 57: Delay process

Ty

Definition 2.9. Throughput

Throughput is specific to flow or communicating pair. It is the rate at which bits are transferred from src to dest in a given time window $\frac{\text{bits}}{\text{time}}$. It can be either instantaneous (rate at given point in time) or average (rate over longer period of time).

2.3 Internet

Definition 2.10. Protocol

Communicating nodes must agree on certain rules. A protocol define format, order of msgs sent out and received among network entities, and actions taken on msg transmission, receipt. Network protocols can be implemented either as software or as hardware.

Definition 2.11. Packet Switching

The internet uses packet switching technology. Different flows (source-dest pairs) share resources (links) along their routes. Internet traffic is bursty in nature. If one flow is not using a link, then other flows can use it. Flows can change routes if link fails or becomes congested.

Definition 2.12. Circuit Switching

Before the internet, circuit switching was used in telephone networks. A circuit consists of all communication links and nodes along a path from source to destination. In circuit switching, a circuit is reserved for each flow for the entire call duration. Guaranteed rate of communication. Flows do not share resources. If one flow is not using its assigned circuit during the call, it cannot be used by another flow. Not ideal for bursty internet traffic. If link fails, call must end.

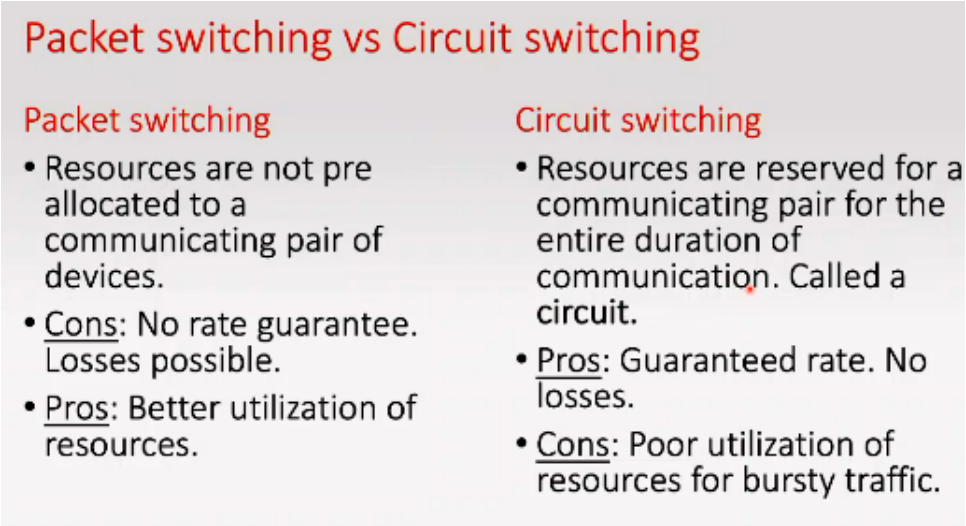


Figure 58: Packet vs Circuit

Definition 2.13. Design Philosophy Layering

Network devices perform complex function. It is better to divide the functions into "layers". Each layer performs a subset of functions. Layer N uses the services of Layer $N-1$ and provides services to Layer $N+1$. Layering makes it easier to add services to a layer or change its implementation without affecting other layers. The internet protocol stack has 5 layers containing all possible functions.

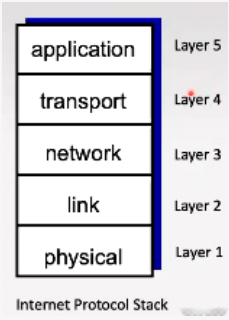


Figure 59: Internet Protocol Stack

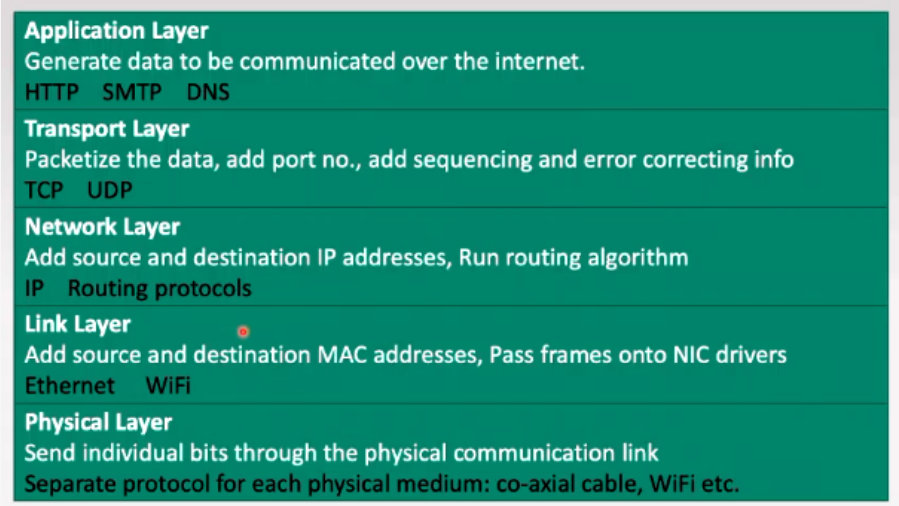


Figure 60: Internet Protocol Stack

Definition 2.14. Sockets

Sockets are APIs between the application and transport layers. It is analogous to doors. Whenever the sender has a message to send, it creates a socket and writes message onto the socket with proper addressing info. The layers below and the internet is responsible for carrying the message to the receiving process. The receiver has another socket into which the message is written. The receiving process simply reads the message from the socket.

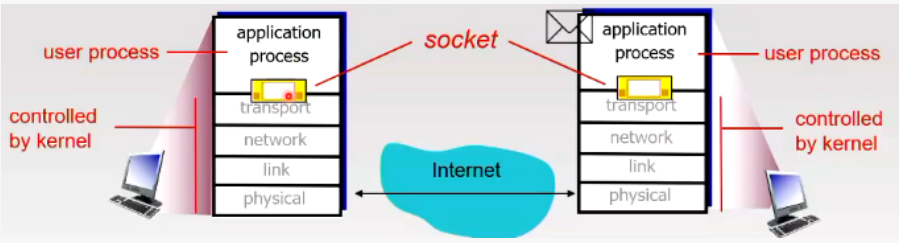


Figure 61: Socket, internet and user process functionality

Definition 2.15. Addressing Processes

Messages need to be addressed to the correct process running within the correct end host. Any internet devices can be identified by its IP address. IPv4 addresses are 32 bit numbers written as dotted decimal e.g., 154.31.16.13. Processes can be identified by port numbers. Port numbers are 16 bit numbers ranging from 0 to $2^{16} - 1$. Port numbers 0 to 1023 are reserved for well known network applications e.g., 80 for HTP. Port numbers above 1023 can be used by other application programmes.

2.4 Transport Layer

Transport layers are used by application processes to deliver messages to their intended recipients. All transport layer protocols offer some basic services. For example, packetisation, addresing, sequencing, error correcting bits. An app may require additional services from the transport layer, such as reliable and in-order delivery of packets.

Definition 2.16. Transmission Control Protocol (TCP)

The TCP offers reliable and in-order data transfer service: packets can be lost or arrive out of order. It also provides connection oriented service: setup required between client and server processes before they start transferring data, called TCP handshake.

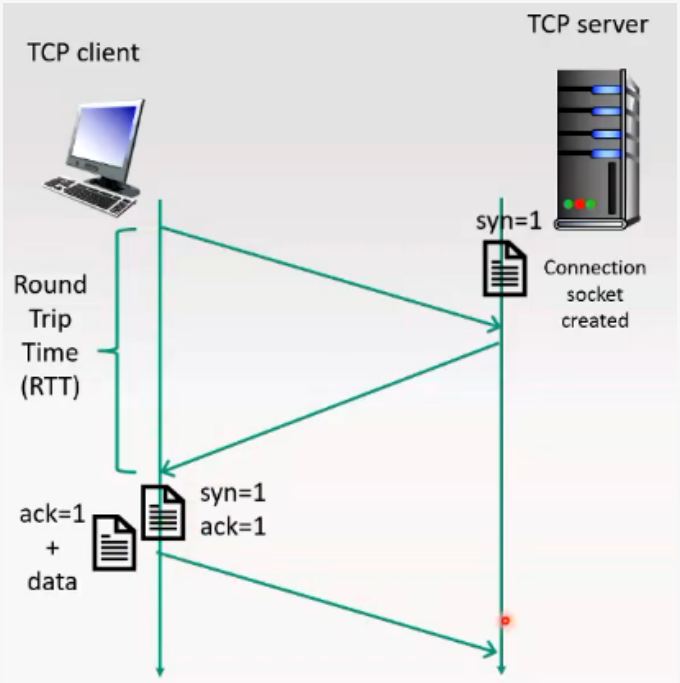


Figure 62: TCP protocol

For example, HTTP uses port 80 and TCP to run. In HTML 1.0, TCP was used separately for every single file found on the server, meaning that there were multiple handshakes causing delay. In 1.1, a single TCP can send multiple files.

TCP also incorporates flow control i.e., it matches the sending speed of the sender to the reading speed of the receiver. It further provides congestion control.

Definition 2.17. User Datagram Protocol (UDP)

UDP provides no guarantees on data transfer. UDP however, is really fast. No connection setup is required, UDP headers are smaller than TCP headers. There is also no effort made to recover losses. UDP is connection-lesss : each UDP segment is treated independently. Furthermore, UDP offers no congestion control. Using UDP, senders can send at any rate they wish even when the network is congested. Despite these, we use UDP mainly because it is fast and it has less overhead. Having no connection establishment (handshaking), meaning that there is less delay. It has smaller header size than TCP. There is no congestion control: UDP sender can blast away as fast as desired such as video streaming, internet telephony etc. Because the transport layer does not have reliability, we instead build the reliability at the application layer.

Example 2.1. Non-Persistent HTTP

1. The client sends SYN, the server responds with SYN-ACK
2. Client sends ACK and adds request for the base HTML file
3. The server establishes the connection and responds with the base HTML file
4. HTTP server closes TCP connection
5. Client receives the HTML file and examines it to find 10 other references objects
6. Steps $1-4$ are repeated for each of the 10 objects

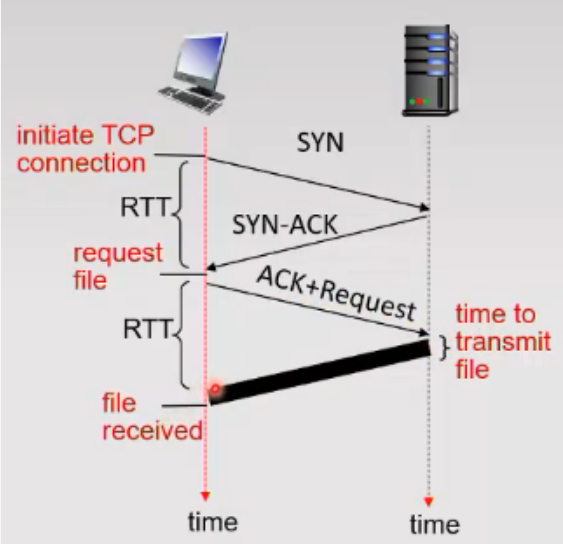


Figure 63: RTT

We use 1 RTT (round trip time) to initiate TCP connection. 1 RTT to send HTTP request and receive first bytes of the requested object. There is also transmission time, meaning

$$\text{Response Time} = 2\text{RTT} + \text{File transmission time}$$

Each object requires at least 2RTT to be downloaded. For each TCP connection, the OS of the server has to allocate some resources e.g., buffers, local variables etc. Problematic when the server has to handle a large number of requests.

Example 2.2. Persistent HTTP

1. The server leaves connection open after sending response
2. Subsequent HTTP messages are exchanged over open connection
3. Client sends request back-to-back as soon as it encounters a references object
4. All objects can be downloaded within 2RTT+ total data transfer time

Definition 2.18. Web Cache

The goal of a web cache is to satisfy client request without involving origin server. Web clients can be configured to access the web via a web cache. Browser sends all HTTP requests to cache. Objects in cache returns the object and if cache request object from origin server, then returns object to client. The idea is that it reduces response time for client request. It also reduces traffic going out of an ISP network. Reduces the cost for the ISP. Lastly, it reduces the traffic on the internet as a whole.

2.5 Selected Topics in Networking

Definition 2.19. Network Interface

A network interface is the point of interconnection between a computer/device and a network. A network interface is typically associated with a hardware network interface card (NIC). Each network in the internet has an IP address. Interfaces associated to hardware NIC also have a hardware address or a MAC address.

Definition 2.20. SYN attack

In a SYN attack, the attacker sends a lot of SYN requests at once. In most cases, they're sent from spoofed IP addresses. The TCP server then responds by creating half-open connections for each request, using a lot of resources. After sending back SYN-ACK packets, the attacker will not respond and the server will have to keep these packets open for a specific period, causing a jam.

Definition 2.21. ARP Cache Poisoning Attack

To understand this attack we need to understand how MAC addresses and what the address resolution protocol (ARP) is.

A MAC changes as a packet travels. For example,

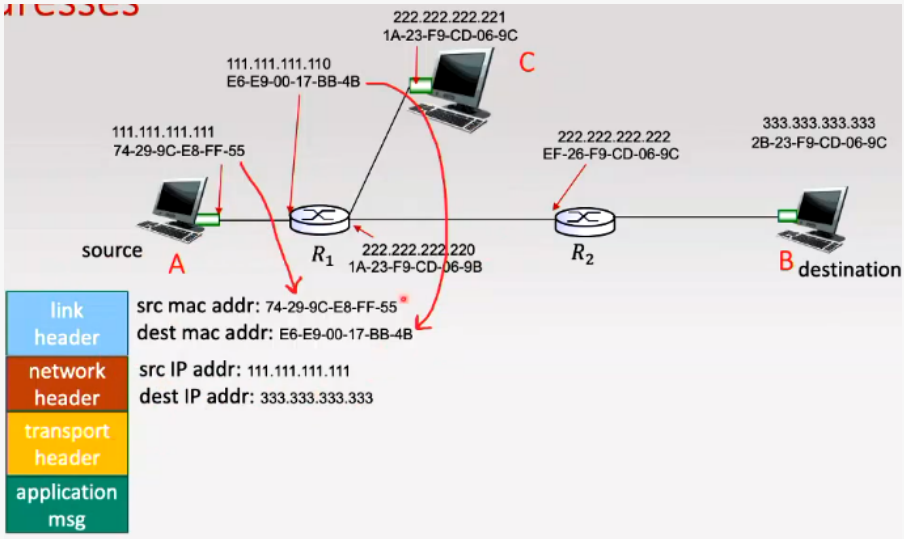


Figure 64: MAC Address in networks

In the above figure, we can see that the network header describes from which IP the packet is sent to, and its destination IP. However, in the link header, we will have a source mac address and destination mac address, which is more local. In other words, if a packet travels from A to B, it first requires to travel from the device to the first router entrance MAC address. Once the first node, the mac address source and destination change once more, i.e.,

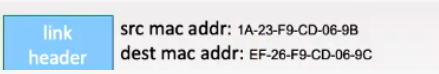


Figure 65: New MAC src and dest

However, we now need to know which new mac address to send it to. This is resolved by the address resolution protocol (ARP). This is done by using the destination IP address. The router will run its own routing algorithm to determine the next destination, specifically, which IP address it should hop into to reach its destination. Once the target is found, the router runs ARP to map the found IP address to a MAC address. The ARP sends an ARP request message to all the devices connected to that router with a specific broadcasting mac address. Only the router with the corresponding IP address will respond to this message. Once the reply is received, it stores the MAC address in the router's cache. ARP allows unsolicited replies since IP addresses can change. An attacn send an unsolicited ARP reply pretending to be someone else.

2.6 Transport Layer 2

Transport layer provides logical communication between app processes running on different hosts. Transport protocols run in end systems. There is the send side, which breaks app messages into segments, adds header and passes to network layer. There is also the receive side, which reassembles segments into messages and passes it to the app layer. See TCP and UDP for a refresh.

Definition 2.22. Reliable transfer over a unreliable channel

We need to understand the general requirements for building reliable data transfer over an unreliable channel. In an unreliable channel there can be bit errors due to electrical noise, loss packets due to buffer overflow and packets can arrive in wrong order due to different delays of channels. Therefore, in the headers we implement

- Checksums - to detect errors. Include the sum of all 16-bit words in the header.
- Acknowledgements or ACKs - to indicate if a packet is correctly received at the receiver.
- Timeout mechanism - sender times out if ACK is not received within a timeout interval
- Retransmissions - to retransmit lost or corrupted packets. Automatic Repeat Request (ARQ)
- Sequence Number - to correctly order packets

Definition 2.23. Stop and Wait ARQ

In stop and wait ARQ, the sender sends a packet and waits until it receives ACK. If ack arrives, sends the next packet else times out and retransmits the same packet.

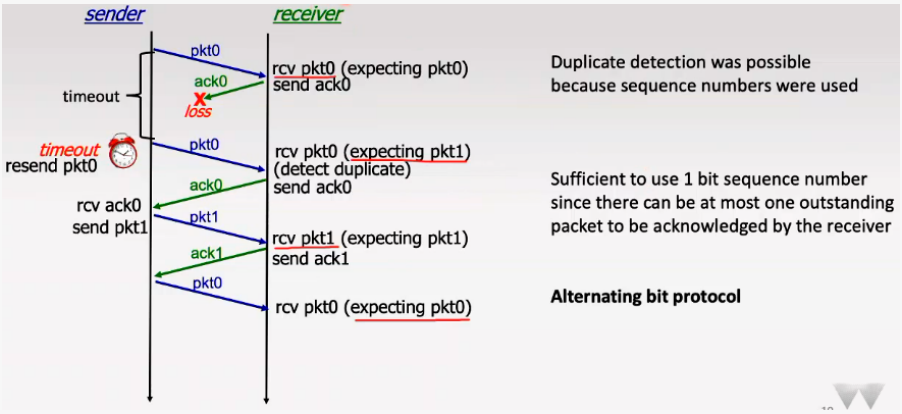


Figure 66: Stop and wait ARQ

Example 2.3. Performance of stop and wait ERQ

Suppose we have a 1 Gbps link $R = 10^9$ bps. Packet length is $L = 8000$ bits.

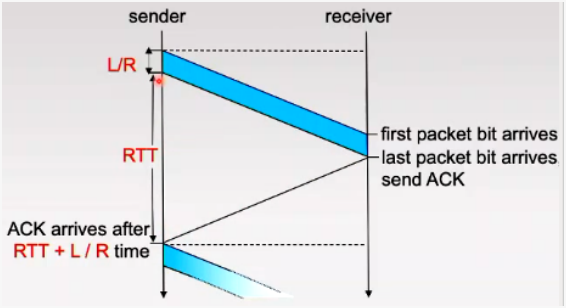


Figure 67: Problem in an image

The utilisation U is then calculated by

$$U = \frac{\frac{L}{R}}{RTT + \frac{L}{R}} = \frac{L}{R \times RTT + L} = 0.00027$$

which, for window based pipelined protocols are

$$U = \frac{min\,(B, L + R \times RTT)}{L + R \times RTT}$$

Definition 2.24. Delay Bandwidth Product

Sender should be allowed to send more packets without waiting for the ACK. Sender could send $R \times RTT$ bits of additional data during the RTT interval. Furthermore, for the length of pipeline defined as $L + R \times RTT$ with buffer size B bits, we have that the maximum number of bits without waiting for ACK is

$$min(B, L + R \times RTT)$$

Definition 2.25. Pipelined protocols

Pipelined protocols allow multiple unacknowledged packets in the pipeline. ACK is sent individually or cumulatively. Range of sequence numbers must be increased. There are two generic protocols called Go-Back-N and Selective Repeat.

Definition 2.26. Go-Back-N Protocol

In the Go-Back-N protocol, the sender can send up to N packets without waiting for ACK. N is the window size, depends on the delay-bandwidth product, receive buffer size. The receiver maintains a variable $expectedseqnum$ which keeps track of the next expected sequence number to be received. If the receiver correctly receives packet n and $n = expectedseqnum$ then it sends $ACK(n)$ which acknowledges all packets up to including packet n. This is called cumulative ack. In the case of $n \neq expectedseqnum$, the receiver discards the incoming packet and sends $ACK(expectedseqnum - 1)$

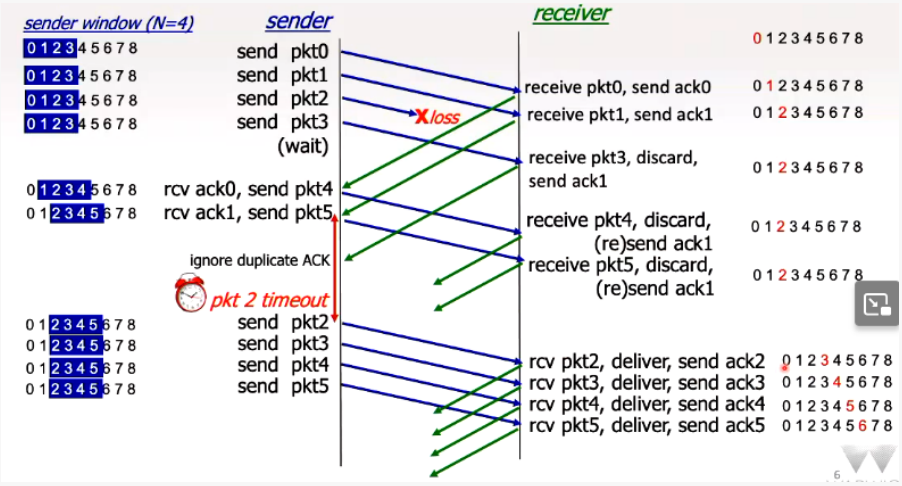


Figure 68: Go-Back-N Visualised

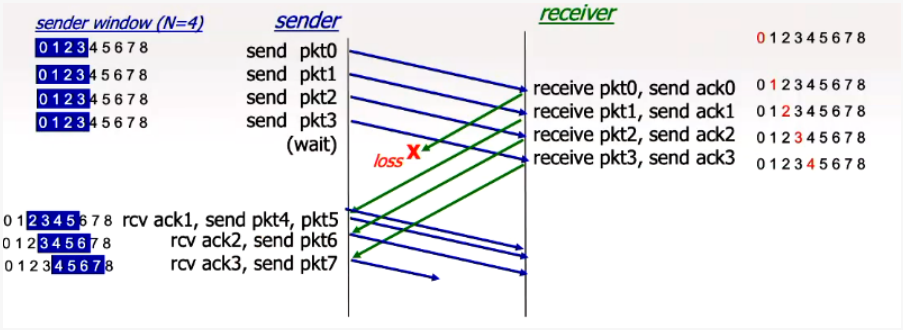


Figure 69: Go-Back-N Scenario 2 Visualised

Definition 2.27. Selective Repeat (SR)

The SR receiver does not discard out of order packets, as long as they fall inside the receive window. The receiver has a window size N which is the same size as the send window N. ACKs are also not cumulative but are individual, meaning that each packet gets acknowledged individually. The sender maintains a timer for each unacked packet in its send window. If the timer runs out, the sender sends that specific timed out packet again. The SR sender does not have to retransmit out-of-order packets.

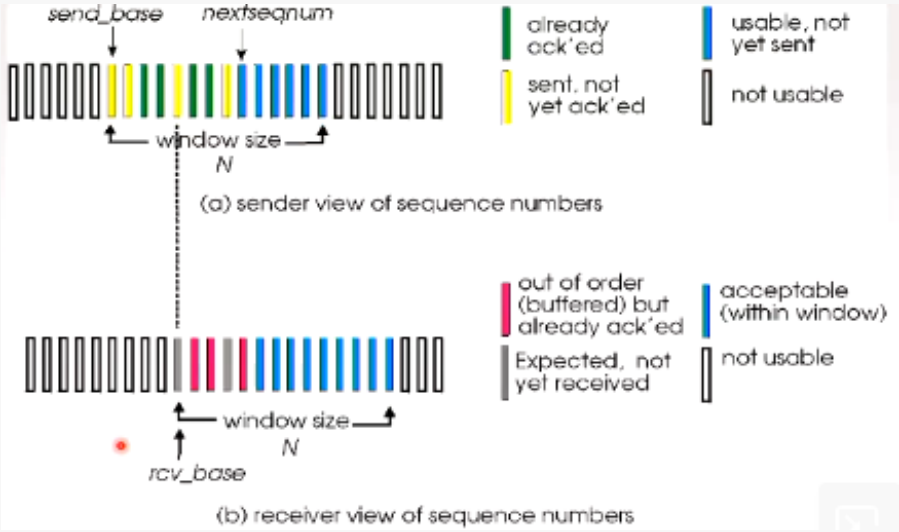


Figure 70: Selective Repeat

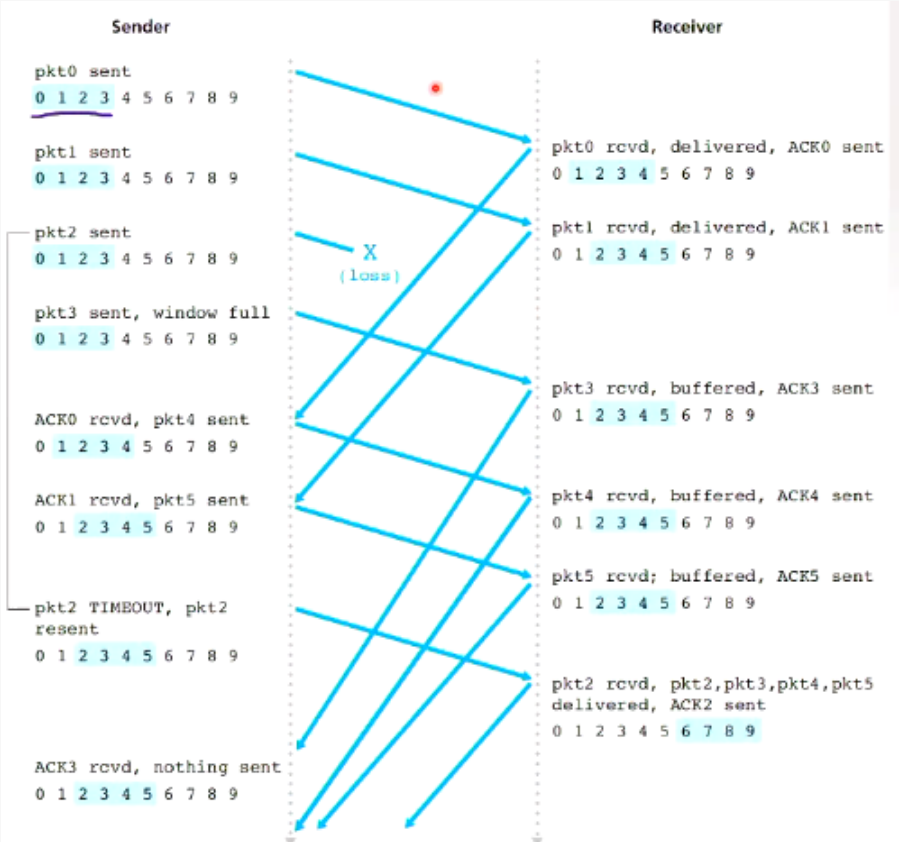


Figure 71: SR in action

Definition 2.28. TCP reliable data transfer

TCP uses a combination of GBN and SR protocols. Like GBN, TCP uses cumulative ACKs and like SR, the TCP sender only retransmits the segment causing timeout. To understand how TCP works, we first need to understand how sequence numbers are assigned to packets. In TCP, each byte of data is numbered. Sequence number of a transmitted TCP segment is the "byte" number of the first byte of the segment. TCP ACK is the number of next expected byte from the other side. TCP uses cumulative ACKs. In TCP, data can flow in both directions. To reduce the number of transmissions, TCP piggybacks ACKs on data segments. A segment can carry data and serve as an ACK.

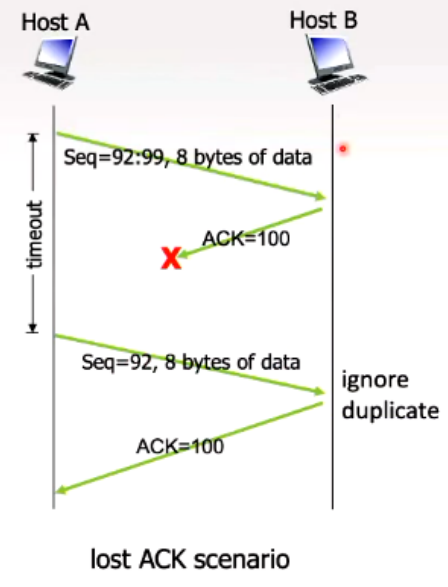


Figure 72: Lost ACK Scenario

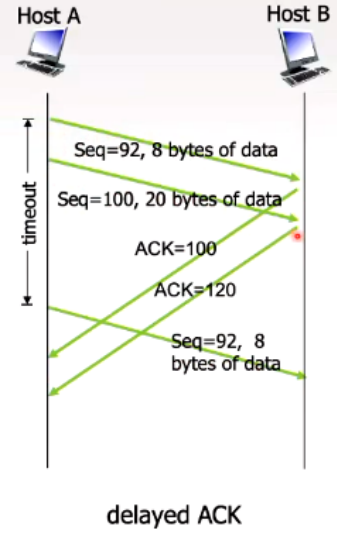


Figure 73: Delayed ACK scenario

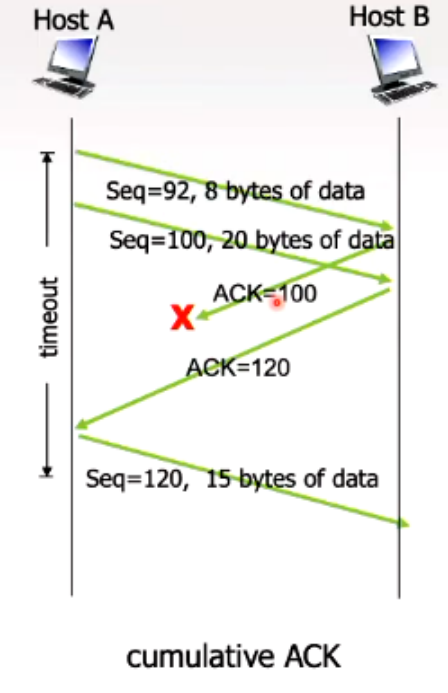


Figure 74: Cumulative ACK Scenario

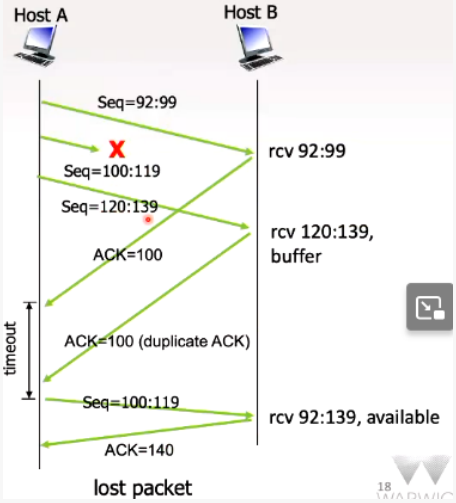


Figure 75: Lost Packet Scenario

The time-out period is often relatively long. Long delay before resending lost packet. TCP fast transmit means that duplicate ACKs are a good indicators of packet loss. They are sent when there is a gap in the received stream of bytes. It allows for faster re-transmit of the lost packet instead of waiting for the timeout period.

Definition 2.29. Flow Control

The data in the pipeline should not exceed the receive buffer size. Otherwise, the receive buffer will overflow and data will be lost. Flow control tries to adjust the sending speed of the sender according to the space available at the buffer. Receiver advertises the free buffer space in the receive window field. It is denoted by rwnd. Sender limits amount of unacked "in-flight" data to receiver's $rwnd$ value.

$$W = LastByteSent - LastByteAcked \leq rwnd$$

This guarantees receive buffer will not overflow

Definition 2.30. Congestion Control

Reliable data transfer and flow control are directly beneficial to the applications. TCP provides congestion control which is beneficial for the network as a whole. It controls the rate of transmission according to the level of perceived congestion.
Congestion at a router occurs when $inputRate > outputRate$. The manifestations include lost packets due to buffer overflow at routers or long delays for queueing in router buffers. The key cause of a congestion in a network are senders sending packets too fast. The goal of congestion control is to control the rate of the senders such that congestion does not occur in the network and that each flow gets a fair share of network resources. A TCP sender assumes the network to be congested when

1. Timeout occurs
2. Three duplicate ACKs are received

TCP treats these events differently. A timeout event is taken more seriously than the reception of 3 duplicate ACKs. If window size is W bytes, the rate of transmission is approximately

$$\frac{W}{RTT}$$

Controlling W controls the transmission rate. The maximum size of a TCP segment is called the Maximum Segment Size (MSS). The number of segments to transmit all data in the window is

$$\left\lfloor \frac{W}{MSS} \right\rfloor$$

Sender maintains the congestion window size denoted by $cwnd$ and

$$W = LastByteSent - LastByteAcked \leq min\,(cwnd, rwnd)$$

$cwnd$ is dynamic, function of perceived network congestion. It is calculated by the idea that the sender linearly increases transmission rate (cwnd) until loss occurs. If loss occurs reduce he rate by a factor of 2. Additive increase: $cwnd$ by 1 MSS every RTT until loss is detected. There is also multiplicative decrease where we cut $cwnd$ in half after loss.

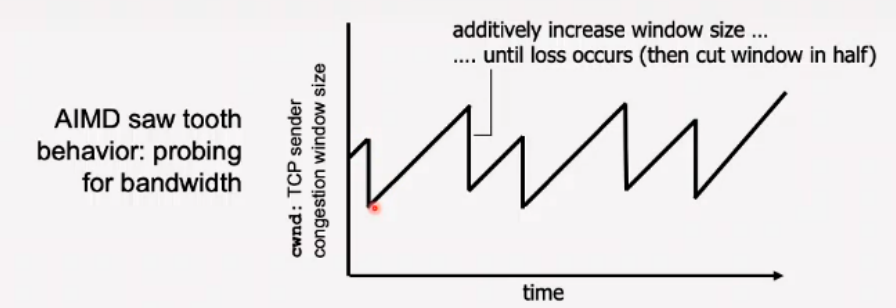


Figure 76: AIMD

This allows fair rate allocation among competing flows.

Definition 2.31. TCP Slow Start and ssthresh

Slow start is when initially $cwnd = 1MSS$, then double each RTT until a loss is detected or slow start threshold (ssthresh) is reached. One ssthresh is reached, we start AIMD, i.e., increase $cwnd$ by $1MSS$ each RTT. ssthresh remembers the previous window size for which a loss occured

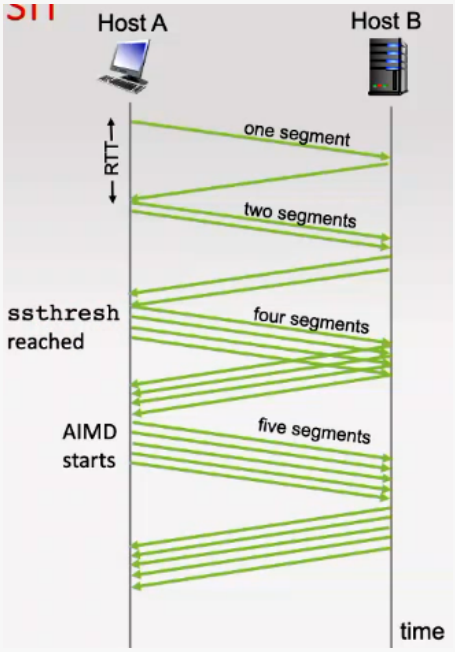


Figure 77: Visualisation of ssthresh

Definition 2.32. Reacting Losses: timeout and duplicate ACKs

Losses are detected through timeouts and 3 duplicate ACKs. Duplicate ACKs is wheen some segments are lost but some are received so we can be lenient. For a timeout, no segment is received so we take drastic measure.
Loss indicated by timeout we take drastic action, we set $ssthresh = 0.5cwnd$ and $cwnd = 1MSS$. After the sender enters into a slow start phase.
When loss is indicated by 3 duplicate ACKs, we take less drastic action. i.e., we set $ssthresh = 0.5cwnd$, $cwnd = \frac{1}{2}cwnd$ and window then grows linearly. In old versions, duplicate ACKs and timeouts were treated the same e.g. TCP tahoe version.

2.7 Network Layer

Definition 2.33. Main Functions

The main function of the network layer is to move packets from the source node to destination node through intermediate nodes called routers. Network layer runs in end hosts and routers. One of the main protocols running in the network layer is IP. IP at source adds IP header, containing src and dest IP addresses, to transport layer segments and sends them to link layer below. The IP at routers checks the destination IP address of incoming packets to decide the next hop router. IP at destination receives the IP datagram, strips IP header and delivers to transport layer. There are other functions of the IP protocol e.g., fragmenting oversized packets and reassembles them at the destination.

Definition 2.34. Two key functions of routers

The two key functions of routers include

- Forwarding - move packets from router's input to appropriate router output
- Routing - construct routing table and use routing protocols

Definition 2.35. Routing Table

Routing table has to store for a 32 bit system, i.e., 2^{32} possible IP addresses. It is impractical to keep a separate entry for each IP. It increases the size of routing table and therefore the look up time. Many IPs map to the same outgoing link. Therefore, we have range of IP addresses stored in the routing table instead.

Definition 2.36. Longest Prefix Matching

When looking for forwarding table entry for given destination address, we use longest address prefix that matches destination address. If there are several entries that match, we choose the address range that has the maximum overlap in numbers.

Definition 2.37. IP addressing and subnets

IPv4 addresses are 32 bit addresses which uniquely identify network interfaces. It is represented in dotted decimal, i.e.,

$$223.1.1.1 = 11011111\ 00000001\ 00000001\ 00000001$$

IP addresses belonging to the same subnet have the same prefix: this is the subnet mask. Interfaces belonging to the same subnet are connected by a link layer switch and can communicate directly with each other. A link layer switch uses MAC addresses to forward link layer frame from source to destination, both belonging to the same subnet. With each IP address its subnet mask is also specified using the number of bits used in the prefix. More specifically,

$$223.1.1.2/24$$

Would denote that the 24 bits is the subnet mask. In other words, the first 24 digits are all the same in the IP address. The notation, Classless Inter-Domain Routing (CIDR) is denoted by

$$a.b.c.d/x$$

where x is the number of bits in the IP address to be used as a subnet mask. A sender first checks if the destination IP has the same prefix as its subnet mask. If so, then we obtain the MAC address of the destination through ARP, create a link layer frame and forward it to the link layer switch.

Definition 2.38. Default Gateway

If source and destination belong to different subnets, then source forwards the packet to its default gateway. A default gateway router connects one subnet to other subnets. If A wants to communicate with B, then A will forward its packets to B, the default gateway.

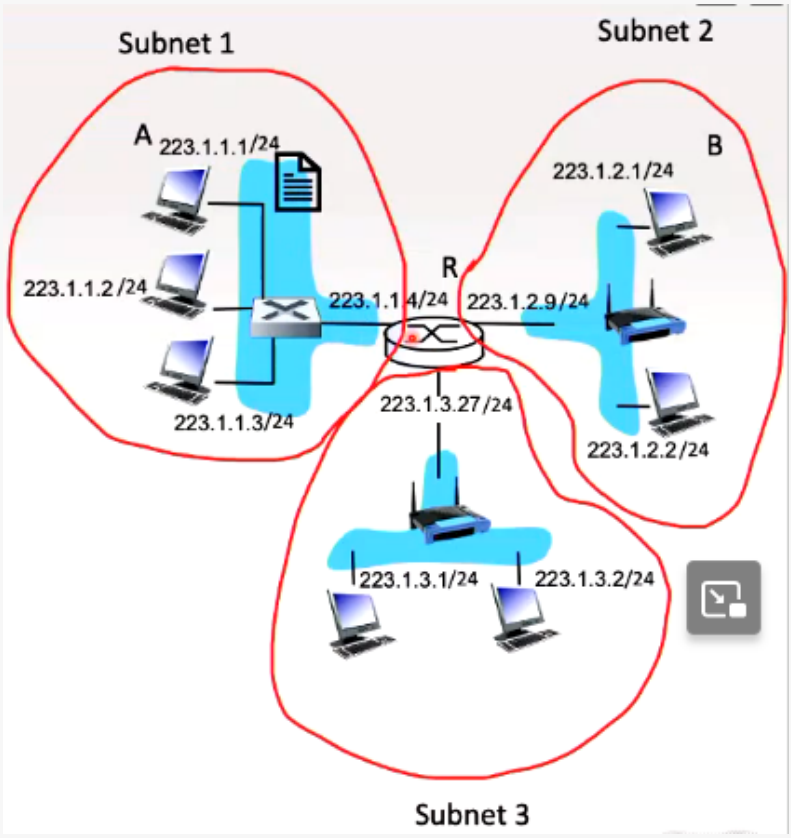


Figure 78: Subnets example

After receiving packet from A, R will lookup its routing table to forward it to the right outgoing interface. Once the packet reaches the interface, it can be forward to B through the switch in subnet 2.

Definition 2.39. How IP addresses are obtained

Network admins can manually configure IP address of each host in a network. UNIX based system store network configuration in a system file e.g., rc.config. If the network is large, it is impossible to get an IP address be assigned manually. Instead, we use the DHCP, that is, the Dynamic Host Configuration Protocol. It is the application layer protocol that dynamically assigns IP address from a server to clients. In either method, subnet mask and default gateway must be specified. But how does network get subnet part of IP address? It gets allocated portion of its provider ISP's address space.

Definition 2.40. Network Address Translation (NAT)

NAT was invented to remedy the idea that IPv4 addresses are short in supply. In fact, ICANN gave out the last block of IPv4 addresses in 2011. It is not possible for ISPs to provide a unique IP to each device connected to a subnet, we instead assign a globally unique public IP address to the gateway routers. As for devices, instead of making them globally unique, we make them privately unique within that specific subnet

Definition 2.41. Public and private IP addresses

Devices in home or private networks need not be visible to the public internet, they can use private IP addresses to communicate with each other. In fact, the addresses

$$10.0.0.0 \text{ to } 10.255.255.255$$
$$172.16.0.0 \text{ to } 172.31.255.255$$
$$192.168.0.0 \text{ to } 192.168.255.255$$

Have been reserved by the ICANN to be used for private IP addresses.

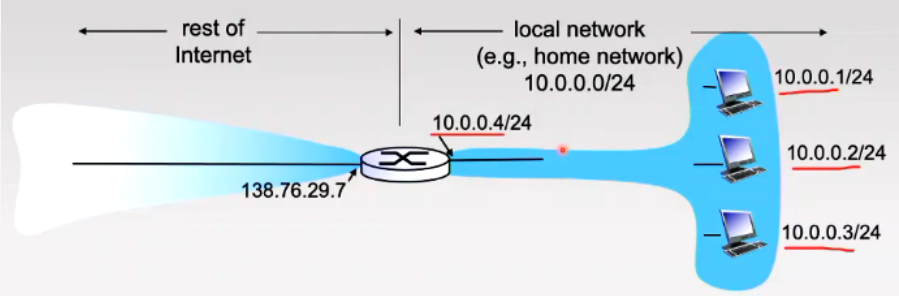


Figure 79: NAT demonstration

However, now we have to distinguish between private hosts when a packet arrives. The network address translation table remedies this problem. It translates the WAN side address to LAN side address. When a packet is requested, it is written in the NAT translation table. When reply arrives, the router can immediately recognise which packet host the packet is meant to send. However, NAT is controversial.

- Port numbers should be used to identify processes not hosts
- Address shortage should instead by solved by IPv6

Definition 2.42. Routing

At each router, a routing protocol e.g. RIP, OSPF constructs the routing table. Each routing protocol implements a routing algorithm. We shall study routing algorithms instead of routing protocols. We will now abstract networks using graph theory, specifically undirected graphs. We will also assign a positive weight to each edge. This cost represents the distance between two nodes or the congestion. The key question then becomes
Given a source x and destination y, what is the least cost path from x to y.
There are two types of algorithm classifications

- Global - requires the knowledge of the complete topology at each router including costs
- Local - requires the knowledge of only local neighbourhood at each router

Theorem 2.2. *Dijkstra's Algorithm*

Computes least cost paths from one node "source" to all other nodes. It is implemented in the Open Shortest Path First (OSPF) protocol. Each node requires the entire topology including the cost of each link. It is obtained through broadcasting of link costs. The pseudocode is as follows

```
Initialization:
   N' = {u}    // set of visited nodes, initially contains only the source
   for all nodes v
    if v adjacent to u:
        D(v) = c(u,v) // stores the current estimate of the shortest distance
        p(v)=u        // stores the predecessor node of v along the current shortest path from u to v
    else D(v) = ∞ , p(v)=NULL
 Loop:
   find w not in N' such that D(w) is a minimum
   add w to N'                    // least cost path to w definitively known
   for all v adjacent to w and not in N' :
     if D(v) > D(w)+c(w,v)  // update distance to the unvisited neighbor v of w
        D(v) = D(w) + c(w,v)    //  if the new distance through w is smaller
        p(v)=w
until all nodes in N'
```

Figure 80: Dijkstra's Algorithm Pseudocode

Example 2.4. Dijkstra's Algorithm

Consider the following graph for an example:

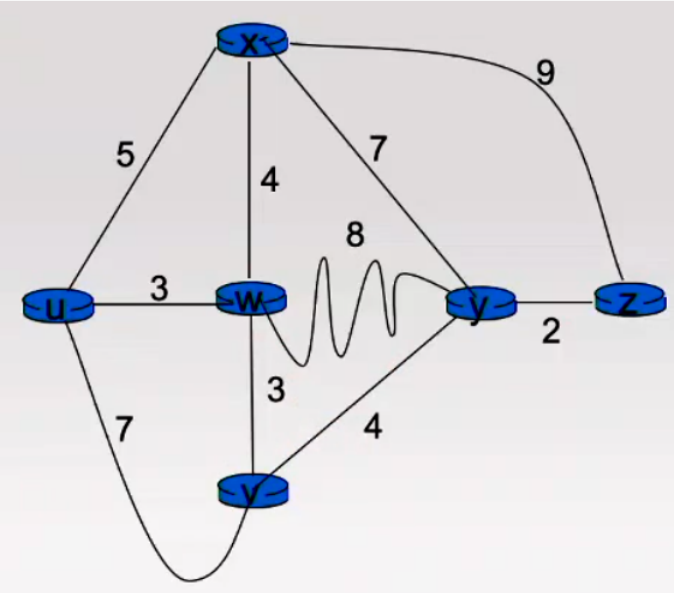


Figure 81: Example Network

We now create a table as following

Table 3: Dijkstra

		$D(v)$	$D(w)$	$D(x)$	$D(y)$	$D(z)$
Step	N'	$p(v)$	$p(w)$	$p(x)$	$p(y)$	$p(z)$
0	u	$7, u$	$3, u$	$5, u$	∞	∞

The above results show our initial estimation from nodes state in columns to the node u. We now select the node with the least value node, that is w of cost 3 and we update the table as follows:

Table 4: Dijkstra

		$D(v)$	$D(w)$	$D(x)$	$D(y)$	$D(z)$
Step	N'	$p(v)$	$p(w)$	$p(x)$	$p(y)$	$p(z)$
0	u	$7, u$	$3, u$	$5, u$	∞	∞
1	uw	$6, w$		$5, u$	$11, w$	∞

Notice that u to v is now a lesser cost from 7 to 6, therefore we have updated the entry in the table for v and y, whilst keeping x. We continue like this for our next least code node, x.

Table 5: Dijkstra

		$D(v)$	$D(w)$	$D(x)$	$D(y)$	$D(z)$
Step	N'	$p(v)$	$p(w)$	$p(x)$	$p(y)$	$p(z)$
0	u	$7, u$	$3, u$	$5, u$	∞	∞
1	uw	$6, w$		$5, u$	$11, w$	∞
2	uwx	$6, w$			$11, w$	$14, x$

And we continue until we obtain the final table as follows

Table 6: Dijkstra

		$D(v)$	$D(w)$	$D(x)$	$D(y)$	$D(z)$
Step	N'	$p(v)$	$p(w)$	$p(x)$	$p(y)$	$p(z)$
0	u	$7, u$	$3, u$	$5, u$	∞	∞
1	uw	$6, w$		$5, u$	$11, w$	∞
2	uwx	$6, w$			$11, w$	$14, x$
3	$uwxv$				$10, v$	$14, x$
4	$uwxvy$					$12, y$
5						

The end result is now a graph explored as such:

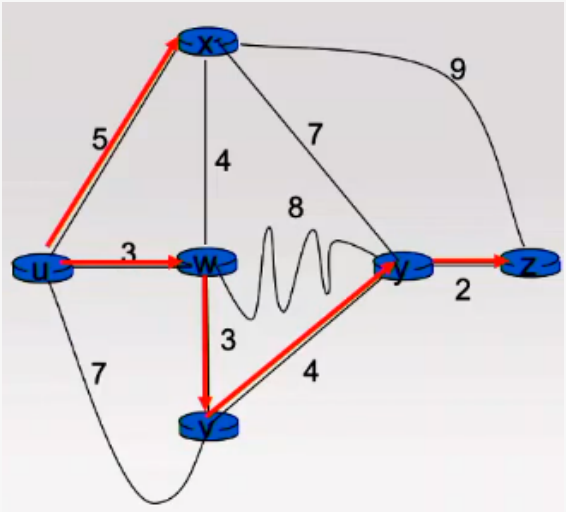


Figure 82: Dijkstra End

Which also creates the shortest path tree.

Theorem 2.3. *Bellman-Ford Equation*

The Bellman-Ford equation denotes
$d_x(y)$ as the length of shortest path from x to y
BF equation relates $d_x(y)$ to $d_v(y)$ where $v \in N(x)$ (set of neighbours of x). The BF equation is as follows:

$$d_x(y) = \min_{v \in N(x)} \{c(x,v) + d_v(y)\} \tag{5}$$

If v^* minimises the above sum, then $v; *$ is the next-hop node in the shortest path.

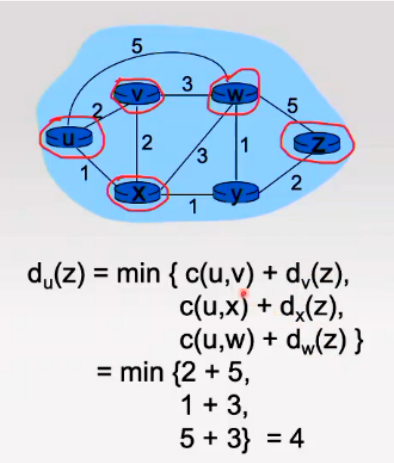


Figure 83: Example of BF

Definition 2.43. Distance Vector Algorithm

The distance vector algorithm keeps the current estimate of minimum distance from x to y. I.e.,

$$D_x(y) = \text{current estimate of minimum distance from } x \text{ to } y$$
$$D_x(y) \neq d_x(y)$$

The DV algorithm tries to converge estimates to their actual values. Each node x maintains distance vector

$$D_x = [D_x(y) : y \in V]$$

Node x performs update $D_x(y) = \min_v \{c(x,v) + D_v(y)\}$
To perform the update node x needs:

- Cost to each neighbour v: $c(x,v)$
- Distance vector of each neighbour v: $D_v = [D_v(y) : y \in N]$ which is obtained through message passing

1. For each node we wait for change in cost or msg from neighbour.
2. Recompute estimates using $B - F$ eqn.
3. If DV to any dest has changed, notify neighbours

Example 2.5. Distance Vector

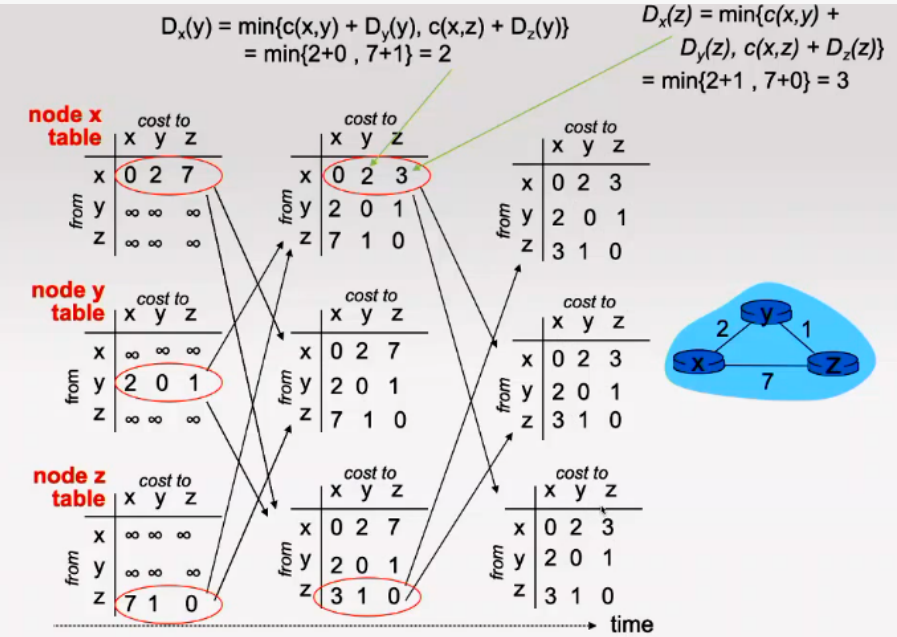


Figure 84: Distance Vector Example