**Define Monolithic and Layered System with suitable examples.**

**Monolithic System (Monolithic System Operating System)**

Imagine a restaurant kitchen where every chef is free to grab ingredients and equipment from anywhere in the kitchen without any strict rules about who should do what. Every chef can call another chef for help at any time.

**Key Components:**

1. **Main Program (The Head Chef)**: The head chef (the manager) oversees the entire kitchen. When an order comes in, the head chef assigns tasks to the chefs (procedures) and oversees everything.
   * **Real-life example**: The head chef receives a customer order (like making a pizza) and assigns various tasks (slicing vegetables, making dough) to the chefs, but there's no clear separation of who should do which task, and everyone can help with everything.
2. **Service Procedures (The Chefs)**: These are the chefs who do the actual cooking. Each chef can work independently but is free to help any other chef who is struggling or needs extra hands.
   * **Real-life example**: A chef working on making pizza dough can suddenly switch to making sauce or chopping vegetables if they need to. There’s no strict rule on who does what.
3. **Utility Procedures (The Assistants)**: These are like kitchen helpers who make sure the chefs have everything they need, like cleaning dishes, organizing ingredients, and preparing tools.
   * **Real-life example**: These assistants bring the ingredients to the chefs, clean the tools, and ensure everything is in place for smooth cooking. They aren’t directly cooking, but they are important.

**How It’s Put Together:**

* **Compilation & Linking (The Process of Combining Everything)**: After all the chefs have prepared their dishes, everything is combined to form a full meal. The chefs themselves may have some overlap in their roles but can step in wherever needed.
  + **Real-life example**: At the end of the shift, all the dishes from each chef are served together as a complete meal, with little regard for how the individual tasks were divided at the start.

**Layered System (Layered System Operating System)**

Now imagine a corporate office where the staff works in distinct layers, and each layer has a different responsibility. The office structure has a clear hierarchy, and each department only communicates with adjacent departments (layers).

**Layers in the Office System:**

1. **Layer 1 (Top Management)**: The top executives (CEO, managers) make high-level decisions and manage the overall direction of the company.
   * **Real-life example**: Think of the CEO deciding on the company’s goals for the year. They don’t get involved in the day-to-day work but oversee the company's strategy.
2. **Layer 2 (Human Resources)**: The HR department handles hiring, employee benefits, and other personnel-related activities.
   * **Real-life example**: HR handles employee contracts, pays salaries, and manages employee relations. They don’t interfere with the work of the finance department, but they make sure the employees are taken care of.
3. **Layer 3 (Finance)**: The finance department manages the company’s budgets, investments, and expenses.
   * **Real-life example**: The finance team processes invoices, tracks revenue, and makes decisions on where to allocate funds, but they work closely with HR to ensure employees are paid correctly.
4. **Layer 4 (IT/Technology Support)**: The IT team is responsible for maintaining the company’s tech infrastructure, like computers and servers.
   * **Real-life example**: The IT department ensures the company’s computer systems are working, fixes technical problems, and implements new software, but they don’t handle human resources or finances.
5. **Layer 5 (Sales/Marketing)**: The sales and marketing teams handle client relations and advertising for the company’s products or services.
   * **Real-life example**: The sales team sells products, while the marketing team works on advertisements. They don’t need to directly interact with the IT or HR departments unless there’s a specific need.
6. **Layer 6 (Operations/Workers)**: These are the workers who execute the daily tasks and operations, such as customer support or product assembly.
   * **Real-life example**: The employees on the ground handle customer inquiries or manufacture the products. They don’t get involved in the high-level decision-making, but they follow the directions set by the other layers.

**How It Works:**

* **Clear Separation**: Each department (layer) in the office can only talk to the adjacent department. HR interacts with finance for salary-related matters, but HR doesn’t directly interact with IT unless there’s a technical issue with employee computers.
  + **Real-life example**: The office’s HR doesn’t make technical decisions, and IT doesn’t handle human resources. However, when a new employee joins, HR and IT will collaborate to set up the employee’s computer system, but otherwise, they stay in their respective lanes.

**Comparison:**

* **Monolithic System** is like the kitchen where everyone does everything. There’s no clear distinction between the roles, and everyone can help everyone else. This can lead to chaos or mistakes but can also be fast and efficient in a simple setup.
* **Layered System** is like an office with clear hierarchies and specialized roles. Each department has its own responsibilities, but they cooperate when necessary. This structure is more organized, easier to manage, and allows for growth, but may be slower due to multiple steps and levels of communication.

**Conclusion:**

* **Monolithic System**: Like a kitchen where everything is mixed together and people do whatever is needed, leading to a faster but messier process.
* **Layered System**: Like a structured office with different departments where each layer has its own role, leading to more clarity and easier maintenance but potentially slower response times.

**Batch Operating System**

Imagine you are the **manager of a factory** where workers have to perform a task, like assembling toy parts.

1. **Without a batch system**: If you were managing the factory **one job at a time**, you would:
   * Ask each worker to finish assembling a toy.
   * Check the progress after each toy is completed.
   * If something goes wrong, you would fix it and then move to the next toy.

This would involve a lot of **back and forth** between you and each worker while they work on each toy, checking, fixing, etc.

1. **With a batch system**: Instead of checking on each toy one by one, you decide to **group all the toys together** in batches:
   * You give each worker a batch of toys to assemble.
   * The workers finish the whole batch before they stop.
   * You only check the batch after all toys are assembled.

This way, you are not stopping the workers to check every toy; you are **leaving them to finish the whole batch before you interact with the result**. This makes the process faster because you’re not interrupting the workers.

**Now, applying this to a batch operating system:**

* **Jobs in a batch** are similar to the toys in the factory. Instead of the computer checking and interacting with each job as it runs (like a worker assembling each toy one by one), the computer **runs all jobs in the batch without stopping** to interact with them.
* Once the **batch is completed**, the computer can then check if any jobs have issues or need attention (just like checking the batch of assembled toys after they're all finished).

**Payroll Processing in a Company**

Imagine you work at a company with hundreds or thousands of employees, and your job is to process the employee salaries (payroll) at the end of each month.

**Without a Batch System:**

1. **Individual Processing**:
   * Every time an employee needs to get paid, you would have to **calculate their salary individually**.
   * You’d go through each employee’s details one by one, checking:
     + How many hours they worked
     + Any overtime
     + Deductions for taxes
     + Bonuses, etc.
2. **Time-consuming**:
   * This process is **slow** because you're handling each employee’s payroll separately.

**With a Batch System:**

Now, imagine the company adopts a **batch system** for payroll. Instead of processing each employee’s salary one by one, the company **groups all payroll tasks into one big batch** and processes them all at once.

**How it Works:**

1. **Grouping Jobs**:
   * All the payroll tasks for the month are **collected into a batch**.
   * This includes gathering the data for each employee:
     + Working hours
     + Overtime
     + Deductions
     + Bonuses
   * Instead of dealing with each employee’s paycheck separately, **all this information is gathered into one group or set**.
2. **Processing the Batch**:  
   * Once the payroll information is grouped, the **batch system** processes all the **calculations and payments together**.
   * The system runs through the batch, calculating everyone’s salary based on pre-defined rules, such as:
     + Salary rate
     + Hours worked
     + Deductions
   * **All the calculations are done automatically** for the entire batch, rather than checking each individual’s details manually.

### ****Conclusion****:

In a **batch system**, when many users (like employees in a company) need something done (such as payroll processing), **all the tasks are grouped together and processed at once**. This is **much faster and more efficient** than handling each task one by one.

### What is a Batch Operating System?

Imagine you need to do several tasks, but you can't do them one by one because it would take too much time. So, you group all the tasks together and do them at once in a set order.

In **a batch operating system**, the computer does something similar: it groups several jobs (tasks) together into a "batch," and then processes all the jobs in the batch one after the other, without interacting with each job individually while they are running. The jobs are processed in batches, and the computer moves on to the next batch only when the current one is finished.

### Example: The Postal System

Imagine a **postal worker**:

1. **Grouping jobs**: The postal worker gets several packages, and instead of delivering each package right away, they sort them based on where they need to go (e.g., all packages going to New York are sorted together, all going to Chicago are sorted together).
2. **Processing batches**: Once sorted, the postal worker delivers the packages in batches, one batch at a time. They don't deliver one package, then another, then another. Instead, they take the whole New York batch and deliver it, then take the Chicago batch and deliver it, and so on.

**without direct user interaction =>**

**No direct user interaction**: Unlike other systems (like interactive systems), the batch operating system doesn't allow the user to interact with the system while jobs are being processed. For example, you can't stop a job in the middle or give input to it once it starts. All jobs are processed automatically without real-time input from users.

**No direct job interaction**: The system doesn't interrupt or modify individual jobs during execution. Instead, it processes a batch of jobs at once, and each job in the batch is run without any changes or interference during processing.

**Time-Sharing Operating System (TSOS)**

**1. How It Works – Imagine a Bank Teller Counter**

In a bank, there’s only one teller, but many customers are waiting in line. The teller spends a small amount of time with each customer—just enough to process a single transaction (like depositing cash or checking an account balance). After handling one quick task for one customer, the teller moves to the next person in line, and so on, quickly rotating through all customers.

* **Real-life parallel**: The bank teller quickly switches from one customer to the next, just like the CPU in a time-sharing OS switches between users.
* **Result**: Each customer feels served fairly and quickly because the teller moves efficiently between them.

**2. Key Features – Multi-User Access with Limited Resources**

Let’s say you’re in a coffee shop with free Wi-Fi, and many people are connected at once. The router provides internet in short bursts to each person so everyone gets a turn to download or upload data without slowing down too much.

* **Real-life parallel**: The Wi-Fi router divides internet time among everyone connected, so each person gets a fair share of speed. Similarly, a time-sharing OS lets many users use the CPU without hogging it.
* **Result**: Everyone feels like they’re getting a reliable connection, even though the router is shared.

**3. CPU Scheduling – Fast Switching Like a Pizza Delivery Driver**

Imagine a pizza delivery driver with multiple orders to deliver. The driver doesn’t spend too much time with any one delivery location; they simply hand over the pizza and move on to the next stop. By doing so, they can deliver to multiple customers in a short amount of time.

* **Real-life parallel**: The driver is like the CPU, and each delivery stop is a user’s task. The driver (CPU) spends a short “time slice” at each stop (task), quickly moving to the next.
* **Result**: Each customer feels like their delivery arrived in a reasonable time, even though multiple orders were delivered.

**4. Memory Management – Lockers in a Gym**

Imagine a gym with lockers for members to store their belongings. Each locker is separate, ensuring one member’s stuff doesn’t mix with another’s, which keeps things secure and organized.

* **Real-life parallel**: In a time-sharing OS, memory is allocated in isolated sections for each user’s program, like how each member uses a unique locker.
* **Result**: Each user’s data is safe, and there’s no interference, so programs don’t accidentally overwrite each other.

**5. Virtual Memory – Overflow Parking at a Concert**

At a large concert, the main parking lot fills up quickly, so cars are directed to an overflow lot nearby. Though people have to walk a bit, they’re still able to attend the concert without needing a parking space right by the venue.

* **Real-life parallel**: When memory fills up in a time-sharing OS, the system uses “virtual memory” (overflow space on the hard drive) to temporarily store parts of programs.
* **Result**: The system can run programs even when memory is tight, much like how people still get parking even when the main lot is full.

**6. Pros – Fair Access & Reduced Duplication**

Imagine a public library where people come to use the computers. There’s no need for each person to bring their own laptop—everyone can use the library computers. The library has a system that limits each session to an hour to ensure everyone gets a chance to use a computer without waiting too long.

* **Real-life parallel**: In a time-sharing OS, each user shares the CPU, but with fair limits on time to prevent any one person from monopolizing resources.
* **Result**: People feel it’s a fair system, and it’s cost-effective because they don’t need to bring their own devices.

**7. Cons – Complexity, Security, and Communication Delays**

Consider an event organizer coordinating between multiple vendors for a large festival. The organizer needs to make sure all vendors are following guidelines, getting paid, and not interfering with each other’s setups. If even one thing goes wrong, the entire event could be disrupted.

* **Real-life parallel**: With multiple users on a time-sharing OS, there are risks like security breaches, delays in processing, and the need for strong management to prevent problems.
* **Result**: Managing a system that serves many people at once can be difficult and requires extra safeguards, much like organizing a large event with multiple vendors.

**Summary**

A time-sharing OS is like a service that allows multiple people to share limited resources, like the bank teller, gym lockers, or Wi-Fi. Each user feels attended to fairly and promptly, even though they’re actually sharing the system. However, managing multiple users this way can bring challenges, such as ensuring security, managing memory, and handling high demand, just like any shared public resource.

**How operating is as a resource manager? Explain.**

**CPU Scheduling/Management:**

* **Real-Life Example in Detail:**  
  Imagine a busy restaurant with only one chef. Many orders come in, and each order is a task the chef (CPU) needs to complete. The restaurant manager (the OS) decides which order the chef should work on first. There are different ways the manager can organize this:
  + **Round-Robin Scheduling:**  
    Like a “first come, first served” system. The chef starts working on each dish in the order they arrived, rotating between them to make sure each order gets some progress. Even if one dish takes longer time to cook, the chef moves on to the next dish after a certain amount of time (similar to the OS assigning CPU “time slices” to each process).
  + **Priority Scheduling:**  
    Some orders (like salads) are faster to prepare, so the chef might prioritize those over slow-cooking dishes (like roasts). If a special order comes from the owner, it jumps to the front of the line.
* **How it Relates to the OS:**  
  The OS makes sure every program gets a fair turn with the CPU, so no program uses all the CPU time. It decides which program gets to run and for how long, balancing fairness and priority, just like how the restaurant manager manages the chef’s time.

**2. Memory Management:**

* **Real-Life Example in Detail:**  
  Imagine a hotel with many rooms. When guests (programs) check in, each one is assigned their own room. Each room is isolated so guests don’t interfere with each other’s belongings, just as the OS isolates memory sections for each program to prevent data conflicts.
  + When a guest leaves, the room is cleaned and ready for the next guest. Similarly, when a program finishes, the OS “cleans up” by freeing up the memory space for the next program.
  + If the hotel gets fully booked, they might use overflow rooms or a temporary space to accommodate additional guests. In an OS, this would be similar to virtual memory, where part of the hard drive is used to extend physical memory if RAM is full.
* **How it Relates to the OS:**  
  The OS allocates separate sections of memory to each program to prevent overlap, much like assigning individual rooms in a hotel. It also ensures efficient memory use by freeing up space once a program is done, reducing memory leaks and optimizing performance.

**3. Device Management:**

* **Real-Life Example in Detail:**  
  Imagine a company where several employees need to share devices, like printers, scanners, and projectors. Each employee (representing a program) needs access to these shared resources. The office manager (the OS) controls access by keeping track of which employee is using which device, setting up usage rules, and handling requests:
  + **Device Drivers:**  
    Like each device having a specific “user manual,” device drivers provide a translation layer that allows the computer to communicate correctly with each specific hardware device. If a new printer is added, the office manager has to read its manual to understand how it works—similarly, the OS needs the correct driver.
  + **I/O Requests:**  
    When multiple employees try to print at the same time, the office manager may set up a queue so each person’s print job gets handled in order. The OS does this by managing input/output requests, ensuring devices are accessed in an organized way without conflicts.
* **How it Relates to the OS:**  
  The OS provides a standardized interface for programs to interact with devices, handling driver compatibility and managing I/O requests, much like an office manager coordinating shared equipment usage to avoid overlap or bottlenecks.

**4. File Management:**

* **Real-Life Example in Detail:**  
  Think of a large, organized library where books (representing files) are systematically organized for easy access. Each book is classified by genre, author, or topic, making it easy for readers to find what they need. There are also rules about who can access certain materials:  
    
  + **File Organization:**  
    Just as a library categorizes and shelves books by genre or author, the OS organizes files and folders within the file system so they can be quickly accessed and retrieved. It may use a directory structure (like library sections) to make file locations easy to navigate.
  + **Access Control:**  
    In a library, some books might be restricted to certain readers, requiring permissions or special access. Similarly, the OS controls who can read, write, or modify files by setting permissions, protecting sensitive data from unauthorized users.
  + **File Operations:**  
    The library allows readers to borrow, return, and look up books. Similarly, the OS allows programs and users to open, read, edit, and save files using a standard set of commands, like creating, deleting, and moving files.
* **How it Relates to the OS:**  
  The OS organizes and manages files systematically, just like a library keeps books in order. It manages how files are accessed and ensures only the right people can open or change them, preventing data loss or unauthorized access.

**What is system call? Discuss process of handling system calls briefly.**

**Push Parameters into the Stack**

* Imagine you write down what you need on a note, like “Book ID 123, Read at Table 5, 10 pages.”

**Call Library Procedure**

* You hand this note to a helper at the library counter, who knows how to handle these types of requests.

**Pass Parameters in Registers**

* The helper organizes the information in a specific way that makes it easy for the librarian to understand. It’s like putting your note details into a special form.

**Switch to Kernel Mode**

* The helper presses a button that opens a door to a back room where only librarians (special staff) are allowed to go.

**Examine the System Call Number**

* The librarian looks at your request and sees it’s a “reading request,” not a request to borrow or return.

**Dispatch to System Call Handler**

* Now that the librarian knows what you need, they go to the correct shelf and get the exact book you asked for.

**Run System Call Handler**

* The librarian brings the book to your table and places it there so you can read.

**Return to Library Procedure**

* The librarian hands things back to the helper, who was the one you originally talked to.

**Return to User Program**

* The helper brings you back to your reading table and says, “Here’s the book you wanted.”

**Increment Stack Pointer (SP)**

* After you’re done reading, you tidy up your area before leaving. This is just like resetting everything to how it was when you started.

**Brief Process of Handling System Calls:**

**Request from User Program**: A user program needs to perform a task that requires operating system privileges (e.g., reading a file).

**Library Procedure Call**: The program may call a library function (e.g., read in C), which translates the request into a system call.

**Switch to Kernel Mode**: The system call triggers a special interrupt, causing the CPU to switch from *user mode* (with limited access) to *kernel mode* (with full access). This switch gives the operating system full control to manage hardware and resources.

**Identify and Dispatch to Handler**: The operating system identifies the system call by checking the specific request and uses a lookup table to find the corresponding system call handler.

**Execute System Call Handler**: The kernel performs the requested operation in kernel mode (e.g., reading data from a disk and storing it in memory).

**Return to User Mode**: Once the system call handler finishes, control is returned to the user program in user mode, and the program can continue its operation with the requested data or result.

**History of Operating System**

**The Early Days of Computing:**

In the 19th century, **Charles Babbage** designed the first mechanical digital computer called the **Analytical Engine**. This machine didn't have an operating system, but Babbage realized that it needed software to work. **Ada Lovelace**, who worked with him, is considered the first computer programmer because she wrote the first algorithm for this machine.

**First Generation of Computers (1945-1955):**

* **Hardware**: The first computers were built using **vacuum tubes**, which made them very large, heavy, and consumed a lot of power.
* **Software**: There were no operating systems at this time. Computers were programmed directly in **machine code** (the basic language the computer understands), and everything was done manually.
* **Programming**: A single team of people would design, build, program, operate, and maintain each machine.
* **Example Machines**: ENIAC and UNIVAC were some of the first computers from this generation.

**Second Generation of Computers (1955-1965):**

* **Hardware**: In this generation, **transistors** replaced vacuum tubes. This made computers smaller, more reliable, and less power-hungry.
* **Software**: The first **operating systems** started to appear. **Batch processing** was introduced, where jobs were grouped together and run one after another without user interaction.
* **Programming**: Computers were programmed using languages like **FORTRAN** and **assembly language**.
* **Example Machines**: IBM 7090 and IBM 1401 were some of the popular computers in this generation.

**Third Generation of Computers (1965-1980):**

* **Hardware**: **Integrated Circuits (ICs)** were introduced, making computers even smaller and more efficient.
* **Software**: **Multiprogramming** was developed, which allowed multiple programs to run at the same time. This kept the computer’s processor busy and improved performance.
* **Operating Systems**: Important operating systems of this time included **OS/360**, **MULTICS**, and **CTSS** (a time-sharing system where multiple users could access the computer at the same time).
* **Programming**: More advanced programming languages like **COBOL**, **ALGOL**, and **LISP** became common.
* **Example Machines**: The **IBM 360** and **PDP-11** were key machines in this generation.

**Fourth Generation of Computers (1980-Present):**

* **Hardware**: The use of **LSI (Large Scale Integration)** and **VLSI (Very Large-Scale Integration)** chips made computers much smaller, cheaper, and more powerful.
* **Microcomputers**: Personal computers, like the **IBM PC**, became widely available, changing how people used computers at home and at work.
* **Operating Systems**:
  + **MS-DOS**: Microsoft's first operating system for IBM PCs.
* **Programming**: Programming languages like **C**, **C++**, and **Java** became widely used.
* **Example Machines**: **IBM PCs**, **Apple Macintosh**, and early **Windows-based PCs** are key examples from this generation.

A table with text on it

Description automatically generated

**Define deadlock. How deadlock can be prevented?**

Deadlock is a situation where two or more processes are unable to proceed because each is waiting for the other to release resources.

**Avoidance**

Imagine you are in charge of distributing a limited amount of food at a community event. You have a set number of food packages, and you need to make sure that everyone can get their share without running out of food for anyone else.

Before you hand out a food package to any person, you look at how many packages are left and check if giving one to this person will leave enough for the others who are still waiting. If you give a package to one person, will there be enough for everyone else? If not, you might decide to delay giving out the package or deny the request until there's enough food for everyone.

This process is similar to how the **Banker's algorithm** works. It looks at the current allocation of resources (food, in this case) and checks if giving more resources to a process will lead to a **deadlock**—where no one can get what they need because the resources are all blocked. If giving a resource would cause a deadlock, the request is denied.

**Technical Connection:**

In an operating system, **Avoidance** works the same way. Before a process is granted the resources it needs, the operating system evaluates whether granting the request will cause any **deadlock** situations—where processes are stuck waiting for each other indefinitely. If granting the request could lead to a deadlock, the system **denies** the request.

So, in simpler terms:

* Before giving out resources to processes (like giving out food), the system checks if it will cause a deadlock (like running out of food for everyone else).
* If granting a resource causes a **deadlock risk**, the system **does not** give it out and avoids the issue.

**Mutual Exclusion**

Imagine a bathroom in a small office. There is only one bathroom, and only one person can use it at a time. If two people try to use the bathroom simultaneously, they will cause a problem. Therefore, only **one person** can use the bathroom at a time. This is the concept of **mutual exclusion**.

* **Why it’s important**: The bathroom (like any other resource) is **exclusive** to one person at a time, ensuring that only one person uses it and no conflicts happen.

**Technical Connection:**

In computer systems, **mutual exclusion** refers to the idea that **only one process can access a specific resource at a time**. The resource could be something like:

* A **printer**: Only one person (process) can print at a time.
* A **file**: Only one process can edit the file at a time.
* **Memory space**: One process uses the memory at a time, so other processes can't interfere.

Without mutual exclusion, there could be **conflicts** (just like two people trying to use the bathroom at the same time), which can cause errors or unwanted behavior in the system.

### ****Hold and Wait****:

* **Real-Life Example**: Imagine a person at a coffee shop who wants both a coffee cup and a sugar packet. If the person takes the coffee cup first but then has to wait for the sugar packet, they are holding one resource (the cup) while waiting for the other (the sugar packet).

**How to Avoid Hold and Wait**:  
The person should ask for both the coffee cup and the sugar packet at the same time, so they don’t have to hold one while waiting for the other.

* **Technical Connection**: In an operating system, this concept means that a process (like the person) shouldn’t hold onto one resource while waiting for another. Instead, it should request all the resources it needs at once. This way, it avoids holding onto one and blocking the process.

**No Preemption**

Imagine you are at a checkout counter, trying to buy some items. You attempt to pay using your credit card, but it gets declined because you don’t have enough funds. Now, the cashier asks you to **switch to another payment method**, like paying with cash instead of your card.

Here’s the important part:

* You **release** the card (the resource you were holding) because it’s not working.
* You are **forced** to use cash (another resource) to complete the transaction.

So, the **card** (your resource) is preempted (taken away from you) by the cashier in exchange for **cash** (a different resource).

**Technical Connection:**

In computer systems, **No Preemption** means that if a process (like your payment attempt) is denied a resource (like the card), the operating system forces it to **release** the resources it is already holding.

For example:

* If a process is holding onto a piece of memory but requests more memory and is denied, the system might take back (preempt) the memory the process is already holding and give it to another process.

In other words:

* **No Preemption** ensures that if a process is denied what it needs, it must **let go of other resources** it’s already using, preventing the system from getting stuck (like how you had to switch from your card to cash).

**Circular Wait (Simplified):**

* **Real-Life Example**: Think of four friends who each need a set of tools: a hammer, wrench, screwdriver, and pliers.
  + Friend A has the hammer but needs the wrench.
  + Friend B has the wrench but needs the screwdriver.
  + Friend C has the screwdriver but needs the pliers.
  + Friend D has the pliers but needs the hammer.

If each friend waits for another friend to give them the tool they need, no one can proceed, and they're stuck in a circle, waiting for each other. This is a circular wait.

**How to Avoid Circular Wait**:  
To avoid this, they agree to ask for tools in a specific order: First, the hammer, then the wrench, then the screwdriver, and lastly, the pliers. This way, no one is stuck waiting for something in a circular pattern.

* **Technical Connection**: In a system, if processes can request resources in a set order (like hammer, wrench, screwdriver, pliers), they can avoid getting stuck in a circular wait situation. This ensures that no process is left waiting indefinitely for resources in a cycle.