Lecture Notes for CS347: Operating Systems
Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay

4. Process Synchronization

4.1 Race conditions and Locks

- Multiprogramming and concurrency bring in the problem of race conditions, where multiple processes executing concurrently on shared data may leave the shared data in an undesirable, inconsistent state due to concurrent execution. Note that race conditions can happen even on a single processor system, if processes are context switched out by the scheduler, or are interrupted otherwise, while updating shared data structures.

- Consider a simple example of two threads of a process incrementing a shared variable. Now, if the increments happen in parallel, it is possible that the threads will overwrite each other's result, and the counter will not be incremented twice as expected. That is, a line of code that increments a variable is not *atomic*, and can be executed concurrently by different threads.

- Pieces of code that must be accessed in a mutually exclusive atomic manner by the contending threads are referred to as **critical sections**. Critical sections must be protected with **locks** to guarantee the property of **mutual exclusion**. The code to update a shared counter is a simple example of a critical section. Code that adds a new node to a linked list is another example. A critical section performs a certain operation on a shared data structure, that may temporarily leave the data structure in an inconsistent state in the middle of the operation. Therefore, in order to maintain consistency and preserve the invariants of shared data structures, critical sections must always execute in a mutually exclusive fashion. Locks guarantee that only one contending thread resides in the critical section at any time, ensuring that the shared data is not left in an inconsistent state.

- There have been several solutions to develop software-based locks, but they are cumbersome, and do not work quite well on multiprocessor systems. Note that having a simple `lock` variable and manipulating it will not work, because the race condition will now occur when updating the lock. A better alternative is to use **atomic instructions** that are provided by most CPUs. One such instruction is the `xchg` instruction (the name may differ across architectures). `xchg` swaps the contents of two variables atomically, and returns the old value. For example, `xchg(&lock, 1)` atomically sets the variable `lock` to 1, and returns the old value of `lock`. Such atomic instructions can be used to implement lock and unlock functions. For example, the code `while(xchg(&lock, 1) != 0);` acquires a lock. The while loop returns 1 and continues to run as long as someone else has held a lock, causing `xchg` to return 1. Once the lock is released, `xchg` acquires it atomically and terminates the while loop. Similarly, a lock can be released with `xchg(&lock,0)`.

- Threads that want to access a critical section must try to acquire the lock, and proceed to the critical section only when the lock has been acquired. Now, what should the OS do to a thread when the requested lock is held by someone else? There are two options: the thread could wait busily, constantly polling to check if the lock is available. Or the thread could be made to give up its CPU, go to sleep (i.e., block), and be scheduled again when the lock is available. The former way of locking is usually referred to as a **spinlock**, while the latter is called a regular lock or a **mutex**. (Note that while the term mutex can refer to a generic lock that spins or sleeps, it usually means a lock that sleeps.) The simple locking code based on the `xchg` instruction above spins busily in the while loop, and hence is a simple implementation of a spin lock. A spinlock usually consumes lot more CPU resources than a mutex, and must be used only if the benefit of busy waiting outweighs the cost of a context switch.

- Most operating systems provide one or more mechanisms for locking. The `pthreads` API has support for locks, both spinlocks and regular mutexes. Threads create a shared lock variable, and call the lock/unlock functions of the API while accessing critical sections. The `pthreads` locks are typically implemented using hardware atomic instructions. Some operating systems also implement hybrids between a spinlock and a mutex: the thread spins busily for the lock for a short period of time, or when it expects the thread holding the lock to release it soon, and blocks if the lock isn't released in a reasonable time.

- In addition to implementing APIs for user code, operating systems also implement mechanisms for safely accessing critical sections of kernel code. That is, when a kernel data structure is being manipulated, the OS should ensure that the update to the data structure happens in an atomic manner. On single processor systems, it is enough to disable interrupts when accessing critical sections, and re-enable them after the update is done. Why? Because, unlike user threads that may get context switched out by the kernel, nothing other than interrupts can interrupt the execution of the OS. On multiprocessor systems, however, disabling interrupts alone may not work, because some other process can move to kernel mode on another CPU core and concurrently access kernel code. Note that all processes share the kernel code and context, so multiple processes in kernel mode are in fact executing on the same kernel code and data. In order to prevent concurrent access from another process in kernel mode on another CPU, the Linux kernel primary uses spinlocks to protect its data structures from concurrent updates. Spinlocks are held for small periods of time, so that other kernel threads (on other cores) can wait busily for the lock to be released.

- Any kernel code that holds a spinlock must not do anything that causes the process to block: the code that runs next may require this lock, which will never to released, leading to a **deadlock**. Therefore, spinlocks must be used very carefully. Operating systems also disable interrupts (and any type of preemption or context switches) on the core on which the spinlock is held (not on other cores) while the spinlock is held, to avoid other interrupt handlers from deadlocking over the held lock. (Note that user processes that hold locks can still be preempted by interrupts, as the interrupt handlers execute kernel code and will never deadlock over locks in user code.)

- Threading libraries can provide many other fancy locks, in addition to the simple spinlocks and mutexes. Some examples are given below. However, note that coding using these locks can be reimplemented using the simple lock as a building block, so these variants are more for convenience to programmers than for correctness.

  - When a thread holds a lock, and calls a function that requests the lock again, a deadlock occurs. One way to solve this problem is to have **recursive locks**. A recursive lock can be locked multiple times by the same thread, and every lock operation should see a corresponding call to unlock. A recursive lock is fully unlocked when all lock operations are matched with unlock operations. Note that another thread cannot still lock while the lock is held by one thread. Recursive locks are useful to avoid deadlocks when a lock may be called multiple times by the same thread. But such locks must be used carefully as they no longer guarantee mutual exclusion. APIs like `pthreads` provide recursive locks that can be used in user code.
  - Observe that writing to shared data concurrently leads to race conditions, while multiple threads reading shared data is perfectly fine. Therefore, **read-write locks** can be used to make this distinction. With these locks, there are separate lock functions for reading and writing threads. Multiple threads can be simultaneously granted this lock if all of them request the lock in read mode. However, when a thread tries to lock in write mode, it must wait for other readers or writer to release the lock before it can acquire it. Read-write locks make it easier to allow concurrent reads. Again, most APIs that provide locking functions (mutexes or spinlocks) also provide read/write versions of the locks.

- In addition to mutual exclusion, two other properties are also desirable when several processes must access a critical section. First is **progress**, or **freedom from deadlocks**. That is, if there are multiple processes wanting to access a critical section, one of them should be given access at any point of time. The other property is **bounded wait**, or **freedom from starvation**. Any single process should not be made to wait indefinitely for access to the critical section. Note that the properties of mutual exclusion, progress, and bounded wait are all independent. For example, one can have a solution (trivially: always give access to only one process) that guarantees mutual exclusion and progress, at the risk of starving one or more processes. Locks are generally designed to guarantee mutual exclusion always, but bad implementations may sometimes lead to deadlocks or starvation.

- Below are several guidelines for programmers when using locks.
  - Every piece of shared data (between processes or threads) must be protected by a lock. Note that the kernel only provides mechanisms for locking, but it is up to the user to ensure that the lock is correctly acquired before updating shared data. The kernel does not in any way detect or prevent threads from updating shared data without using a lock, or from holding locks unnecessarily when no shared data is being updated.
  - The user can decide the "granularity" of the lock. The user can use a coarse-grained lock to protect several data structures, and acquire this lock for updates to any of them. Or, the user can use finer-grained locks for different data structures, or even for different parts of one data structure. In general, coarser locks require fewer lock/unlock operations, but prevent concurrent executions. Finer-grained locks impose lock/unlock overheads, but allow more concurrency. The choice of lock granularity depends on the application developer.

  - Note that locking is an expensive operation. Both spinlocks and mutexes incur extra CPU overhead (either due to polling, or due to the time required for sleeping and waking up). Locks also hamper concurrent execution of threads that are contending for a lock. As a result, locks should be used economically within pieces of code. User programs should try to minimize critical sections and work in a lock-free manner as far as possible. Shared data should be partitioned across threads as far as possible so that the need for concurrent updates to the same variable is minimized.
  - Deadlocks can occur when a user program manages multiple locks. For example, if a thread locks A and is waiting for lock B, and another thread locks B and waits for lock A, both threads will deadlock, and neither will make progress. To avoid such deadlocks, user code must avoid holding more than one lock at a time. If multiple locks must be held, the locks must be acquired in the same order across threads.

- An alternative to locks is to use transactional memory: memory that provides atomic updates on variables, eliminating race conditions. The idea of hardware transactional memory has existed for a while. Some new programming languages are also providing software abstractions that realize transactional memory: data is no longer shared, but assigned an "owner". Any thread that wishes to update a variable must send a message to the owner, that will perform the update without worrying about concurrency. For the rest of the course, we will assume shared data and concurrent updates, but this assumption may not always hold in all programming languages in the future.

4.2 Conditional Variables

- Mutual exclusion is only one of the several ways in which processes or threads can synchronize with each other. Another way is **conditional synchronization**, or a process doing something only after another process has done something else. A **conditional variable** is a synchronization primitive that enables this type of coordination between processes. Note that several primitives with several names exist in operating systems literature that provide this functionality (e.g., monitors in Java). However, we will use the term "conditional variable" that is used in the `pthreads` API.

- Conditional variables have several uses. For example, consider a file server process with a pool of worker threads. The file server places incoming requests into a queue, and the worker threads dequeue requests and serve them. The worker threads wait for the condition that the queue has a request, and when a request is enqueued, the waiting worker threads are woken up to proceed.

- Two operations can be called on a conditional variable (or **condvar** for short): a **wait / block / sleep** and a **signal / wakeup**. A process or thread that requires a precondition to be satisfied before it can proceed will call wait on a conditional variable. Another thread that makes this precondition happen will call signal, so that the thread(s) that called wait on this condvar can be woken up. The signal event can wake up one or more waiting threads, depending on the implementation. In addition, the signal to wake up can be broadcast to all waiting threads with a **signal broadcast** function provided by some operating systems.

- The wait/signal operations on a conditional variable must be protected by a spinlock/mutex. A thread calling wait must first lock the mutex, and then call wait, so that the wait can be executed atomically, avoiding race conditions while updating the data structures of waiting processes. The kernel unlocks the mutex after the thread is put to sleep. When the waiting thread is woken up, it returns with the locked mutex once again, which it may choose to unlock. The job of the mutex here is to protect the consistency of the data structures involved in the wait and signal operations, and the mutex is not held during the blocking period.

- What happens if a conditional variable is not put to sleep with a lock? Consider a process P1 that checks a certain condition, finds it to be false, calls wait, and decides to sleep till the condition is true. Now, consider another process P2 that makes the condition true, and calls signal on the condition variable. Due to concurrent execution, suppose that P2 calls signal between P1 checking the condition to be false, and sleeping. In that case, P1 will sleep, even though the condition is true. The wake up call of P2 will not wake P1, as it hadn't slept yet. This will result in a deadlock, with P1 sleeping and never waking up. The operation of checking a condition and sleeping if false should be done atomically, which is the reason a lock must be held when calling wait/sleep. If both processes perform calls to wait and signal only with locks held, the operations are serialized, and the deadlock above would not occur.

- Linux implements conditional variables defined in the `pthreads` API, which can be used by user space processes for conditional synchronization. In addition, Linux implements **wait queues** for a similar functionality for kernel code. For example, when a kernel thread must block on some event (e.g., disk read), it calls the wait/block/sleep function provided by the OS, and the process is added to a wait queue. After the event completes, the waiting thread is woken up (say, by the interrupt handler) by calling signal/wakeup. Wait queue operations are internally protected by a spinlock for atomicity.

- Note that the use of a mutex/spinlock may or may not be apparent when invoking the wait/signal functions on a conditional variable. For example, in the `pthreads` API, a lock must be explicitly specified as an argument to the wait function. However, in Linux wait queues, the spinlock in acquired as part of the implementation of the wait queues itself. Irrespective of whether the lock is apparent or not, some locking mechanism will be used to ensure the atomicity of the wait/signal operations on a conditional variable. Using conditional variables to wait does not completely eliminate busy waiting; but the busy waiting is restricted to a small period of acquiring spinlocks.

- Note that it is wise for users to check that the condition is indeed true when wait returns. Sometimes, the condition may become false once again between the thread waking up and returning to execution, because of some other thread running in the interim. See the code snippets below for clarity.

```
if(condition)
  wait(condvar)
//small chance that condition may be false when wait returns

while(condition)
  wait(condvar)
//condition guaranteed to be true since we check in while-loop
```

4.3 Semaphores

- A lock allows only one thread to access the critical section. On the other hand, a **semaphore** can allow a certain specified number of threads (say, N) to access a critical section. Note that a semaphore is equivalent to a regular lock when $N = 1$. That is, a mutex is a binary semaphore. A semaphore is used to guard a resource that has N instances, allowing access to N concurrent threads. The operations that can be performed on a semaphore variable are called "down" and "up". That is, a thread that wishes to acquire one of the resources guarded by the semaphore will call the down operation (equivalent to locking a mutex). A thread that wishes to release the semaphore calls the up operation (equivalent to unlocking a mutex). When the number of available instances is 0, all further calls to down will not succeed, until some resources are released by the up call. That is, the process calling down will be suspended or blocked, to be woken up later when resources are available. An up operation on a semaphore should not generally block. Note that the down and up operations are also referred to as P and V operations on a semaphore (from Dutch names for the same).

- Note that the up and down operations of a semaphore must be implemented atomically. That is, the steps of checking the current count and deciding to sleep must be performed atomically, otherwise race conditions may result. Further, the count of the semaphore must also be updated atomically. However, not all architectures have atomic hardware instructions that can implement semaphores directly. For such architectures, operating systems use spinlocks to protect the critical sections in the implementations of the up and down operations on a semaphore, and mechanisms like wait queues to implement the sleeping and wakeup actions.

- Below is sample code that implements the up and down operations of a semaphore using mutexes and conditional variables. Note that no kernel code would actually use condvars internally, so the below code snippet is only to give you a high-level idea, and does not correspond to any real implementations. Actual kernel semaphore implementations are more complex.

```
// S is the semaphore variable, S.count is the current count
// lk is a mutex, condvar is a conditional variable

down(S):
  lock(lk)
  while(S.count <=0)
    wait(condvar, lk)
  S.count--
  unlock(lk) //condvar returned with this lock


up(S):
  lock(lk)
  S.count++
  signal(condvar) //wakeup anyone sleeping on down
  unlock(lk)
```

4.4 Examples of synchronization problems

Several process synchronization situations are similar at a high-level, and can be mapped to a small set of classic synchronization problems. Some of these classic problems are described below.

1. **Producers and consumers.** In this problem, one or more processes produce items into a buffer. Another set of processes consume items from the buffer. The buffer can be bounded or unbounded, but let's discuss the bounded case here. Several real life scenarios map to this problem. For example, a disk driver produces disk blocks, and the kernel consumes them and hands them over to waiting processes.

   Several synchronization requirements exist in this problem. Locks have to be used to ensure the integrity of the shared buffer itself, to avoid inconsistent concurrent updates. Producers and consumers may also need to wait for each other: a consumer must wait for the producer to put something in an empty buffer, and a producer must wait for the consumer to take something out of a full buffer. Below is one way of solving the producer-consumer problem using mutexes and conditional variables, like the ones in `pthreads`. Two conditional variables, `buffer_has_space` and `buffer_has_data`, are used, along with an associated mutex.

   Producer:

   ```
   lock(mutex)
   while(itemCount >= MAX)
     wait(buffer_has_space)
   add item to buffer
   itemCount++
   signal(buffer_has_data)
   unlock(mutex)
   ```

   Consumer:

   ```
   lock(mutex)
   while(itemCount == 0)
     wait(buffer_has_data)
   remove item from buffer to consume
   itemCount--
   signal(buffer_has_space)
   unlock(mutex)
   ```

Below is a solution using semaphores. Two semaphores `emptyCount` (initialized to the size of the buffer) and `fullCount` (initialized to 0) are used to indicate the counts of free and full slots in the buffer respectively. Notice how the counts of buffer slots neatly maps to the count of the semaphore, making semaphores a natural technique to solve this problem. However, another mutex/binary semaphore must be used to guard access to the shared buffer itself, when writing to it while producing/consuming.

Producer:

```
down(emptyCount)
lock(mutex)
  add item to buffer
unlock(mutex)
up(fullCount)
```

Consumer:

```
down(fullCount)
lock(mutex)
  consume item
unlock(mutex)
up(emptyCount)
```

Note that most synchronization problems can be solved either with semaphores or with conditional variables; the two mechanisms are equivalent in some sense. That is, the wait action on a condvar roughly corresponds to a down action on a semaphore, while the signal corresponds to the up. When developing a solution to a synchronization problem using any of these techniques, carefully ensure that all calls to wait/down are matched by a corresponding call to signal/up, to ensure correctness of your solution.

2. **Readers and writers.** Several processes wish to read and write data shared between them. Some processes only want to read, while others want to update the shared data. Multiple readers may concurrently access the data. However, a writer must not access the data concurrently with anyone else, either a reader or a writer. Reader-writer locks exist precisely to address such problems. Below is a sample implementation of read/write lock/unlock functions using simple mutexes and conditional variables. A boolean variable `writer_present`, and two conditional variables, `reader_can_enter` and `writer_can_enter`, are used.

   Read lock:

   ```
   lock(mutex)
   while(writer_present)
     wait(reader_can_enter)
   read_count++
   unlock(mutex)
   ```

   Read unlock:

   ```
   lock(mutex)
   read_count--
   if(read_count==0)
     signal(writer_can_enter)
   unlock(mutex)
   ```

   Write lock:

   ```
   lock(mutex)
   while(read_count > 0 || writer_present)
     wait(writer_can_enter)
   writer_present = true
   unlock(mutex)
   ```

   Write unlock:

   ```
   lock(mutex)
   writer_present = false
   signal(writer_can_enter)
   signal(reader_can_enter)
   unlock(mutex)
   ```

3. **The Dining Philosophers problem.** N philosophers are sitting around a table with N forks between them. Each philosopher must pick up both forks on her left and right before she can start eating. If each philosopher first picks the fork on her left (or right), then all will deadlock while waiting for the other fork. The goal is to come up with an algorithm that lets all philosophers eat, without deadlock or starvation. The solution below comes up with a deadlock-free solution, by making a philosopher pick up forks and eat only when both are available.

   A variable `state` is associated with each philosopher, and can be one of EATING (holding both forks) or THINKING (when not eating). Further, a conditional variable is associated with each philosopher to make them sleep and wake them up when needed. Each philosopher must call the `pickup` function before eating, and `putdown` function when done. Both these functions use a mutex to change states only when both forks are available.

```
bothForksFree(i):
return (state[leftNbr(i)] != EATING &&
        state[rightNbr(i)] != EATING)

pickup(i):
  lock(mutex)
  while(!bothForksFree(i))
      wait(condvar[i])
  state[i] = EATING
  unlock(mutex)

putdown(i):
  lock(mutex)
  state[i] = THINKING
  if(bothForksFree(leftNbr(i)))
      signal(leftNbr(i))
  if(bothForksFree(rightNbr(i)))
      signal(rightNbr(i))
  unlock(mutex)
```

4.5 Deadlocks

- Ensuring consistency of shared data is only one of the many desirable properties when multiple processes or threads are working together. Other properties of interest are progress (freedom from starvation) and bounded waiting (freedom from deadlocks). However, none of the synchronization mechanisms studied so far guarantee freedom from deadlocks or starvation. Linux does not provide any explicit mechanisms to avoid deadlocks and starvation, and it is left up to the programmers to write code in a way that it does not deadlock or lead to starvation. For example, when multiple locks are being acquired by processes, it is recommended that the locks be acquired in the same order across processes to avoid deadlocks.

- Processes can deadlock while waiting for any resource (a hardware resource or a software resource such as a lock). A deadlock occurs when the following four (somewhat interdependent) conditions happen together:

  1. *Mutual exclusion.* At least one resource of a process is held in a mutually exclusive fashion.
  2. *Hold and wait.* A process must be holding one resource and waiting to acquire some other resource held by another process.
  3. *No preemption.* A process must be holding a resource in a way that it cannot be preempted to reclaim the resource until the process finishes and gives up the resource voluntarily.
  4. *Circular wait.* This is the most important condition, and is a superset of the conditions above. A set of processes $P_0, P_1, \ldots, P_n$ must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, and so on, and P_n is waiting for a resource held by P_0.

- Deadlocks can be understood with a **resource allocation graph**. The vertices in this graph are processes and resources. Directed edges called assignment edges exist from a resource that is held to the process that holds it. Directed request edges exist from processes to the resources they have requested and waiting for. A directed cycle in the resource allocation graph is a necessary and sufficient condition for the occurrence of deadlocks. Consider the following example: process P1 holds resource A, and is waiting for resource B; process P2 holds B and is waiting for A. We can find that the above deadlocked state corresponds to a cycle in this graph: P1-B-P2-A-P1. (Note that if each resource has multiple instances, then a cycle in the graph is a necessary but not a sufficient condition for deadlocks. We will not discuss the multiple-instance version of the resource allocation graph in much detail.)

- Deadlock *detection* algorithms maintain such resource allocation graphs, and look for cycles in these graphs to identify deadlocks. If a deadlock is detected, one or more deadlocked processes must be terminated and/or resources reclaimed back preemptively, until the deadlock resolves. Recovering from deadlocks is very hard once they occur, so the best way out is to prevent them with careful programming.

- Deadlock *prevention* techniques ensure that at least one of the necessary conditions for a deadlock does not occur, thereby preventing deadlocks altogether. For example, they may require that a process cannot request another resource when holding one resource in a non-preemptive manner already. Or, some techniques impose a total order on resources, and require that processes acquire resources in a certain increasing order, to prevent the circular wait condition. For example, in the example of processes (P1,P2) and resources (A,B) above, if the ordering of resources is (A,B), and if P1 has acquired A and is waiting for B, then P2 must also first acquire A and only then acquire B. This will ensure that P2 never acquires B first, eliminating the possibility of a deadlock. However, deadlock prevention techniques are much too conservative sometimes, and may lead to low resource utilization.

- Another technique is to let all the deadlock conditions coexist, but have algorithms for deadlock *avoidance* in place to constantly check for and guard against the possibility of deadlocks. **Banker's algorithm** is a popular deadlock avoidance algorithm. This algorithm assumes that the following are known: the total number of resource instances (of each type) available in the system, the maximum number of resources required by each process (at any time in its run), and the current allocation of resources to processes. Whenever a new resource allocation request is made, the algorithm first checks if allocating this resource will maintain the system in a "safe" state or not, and allows the request to proceed only if the answer is yes. A safe state is one from which all remaining requests from processes (up to their maximum value) can be satisfied in some order. That is, a *safe sequence* of processes exists such that the processes can satisfy all their requests, finish, and release resources in that order, without deadlocking at any point. If such a safe sequence exists, then the system is guaranteed to not deadlock in the future. However, if such a safe sequence does not exist, then the system is in an unsafe state, and a deadlock *may* occur in the future (if processes do end up requesting their maximum allowed quota). Therefore, by checking that every new request keeps the system in a safe state, the Banker's algorithm avoids deadlocks. This algorithm is generic enough to be used in other domains as well, beyond operating systems. Note that most modern operating systems do not implement any such deadlock avoidance checks, and rely on the user to prevent deadlocks.

- We now provide a formal description of the Banker's algorithm. Let N denote the number of processes (P_i), and M denote the number of resource types (R_j). The inputs to the bankers algorithm are:

    1. Available[M] denotes the current available number of resource instances for each resource type.
    2. Max[N][M] denotes the maximum instances required by any process for each resource type.
    3. Allocation[N][M] denotes the current allocation of resource instances to each process.

- We use the notation $A < B$ for two vectors if each element of A is less than the corresponding element in B. We first describe how to check if the system is a *safe state* or not.

    1. Let CurrentAvlbl[M] = Available, and Finish[N] is a vector initialized to false. We start with the currently available resources, and try to find a safe order in which processes can request their maximum and finish.
    2. Find an index i (i.e., process P_i) such that Finish[i] is false and Max[i] – Allocation[i] $\leq$ CurrentAvlbl. That is, we find a process which can be granted resources up to its maximum value, with whatever resources are currently left. If no such i exists, go to step 4.
    3. Do CurrentAvlbl = CurrentAvlbl + Allocation[i] and Finish[i] = true, to reflect the fact that process P_i has finished and returned its resources back (the algorithm assumes that a process that has received its maximum resources will eventually finish and release the resources). Go to step 2 and repeat.
    4. If Finish is true for all processes, then the system is in a safe state.

- Let Request_i be a vector of resource requests from process P_i. We now check if this request can be accepted or not.

    1. Check that $\text{Request}_i \leq$ Max[i] – Allocation[i] (process respects its maximum specifications) and $\text{Request}_i \leq$ Available (the requested resources are indeed available). If either of these checks fail, the request obviously cannot be satisfied.
    2. If the above checks pass, we pretend that we satisfy the request: Available = Available – Request_i and Allocation[i] = Allocation[i] + Request_i. Now, we check if the system is in a safe state. If it is, we accept the request. Otherwise, we deny the request and roll back the changes made to allocations in this step. If the request is denied, the process must wait to acquire the resources.