Multiple Choice: (1 mark/ question)

- Dining phylosopher

Dining philosophers with semaphores

```
#define N 5 /* number of philosophers */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY 1 /* philosopher is trying to get forks */
#define EATING 2 /* philosopher is eating */

int state[N]; /* each philosophers is either THINKING, HUNGRY or EATING, initialized to THINKING */
sem_t cs_m; /* semaphore for the critical section, shared by all philosophers, initialized to 1 */
sem_t p_m[N]; /* one semaphore per philosopher, initialized to 0 */

int left(i) { return (i+N-1) % N; }
int right(i) { return (i+1) % N; }

void philosopher(int i) /* to be executed by different threads */
{
  while (1) {
    think( ); /* philosopher is thinking */
    take_forks(i); /* acquire two forks or block */
    eat( ); /* yum-yum, spaghetti */
    put_forks(i); /* put both forks back on table */
  }
}
```

```
void take_forks(int i) {
  down(& cs_m);           /* enter critical region */
    state[i] = HUNGRY;    /* record fact that philosopher i is hungry */
    test(i);              /* try to acquire 2 forks */
  up(& cs_m);             /* exit critical region */
  down(& p_m[i]);         /* block if forks were not acquired */
}

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

void test(i) {            /* i: philosopher number, from 0 to N-1 */
  if (state[i] == HUNGRY && state[left(i)] != EATING && state[right(i)] != EATING) {
    state[i] = EATING;    /* only start eating if hungry, and neighbors not eating */
    up(& p_m[i]);
  }
}
```

5 processes/threads, each needs two resources, such as files, to proceed how to allocate resources so that all process/threads get to execute?

Attempt 1: repeat forever: (grab left fork ; grab right fork; eat; put forks back; think)

this would lead to a **deadlock:** assuming all philosophers are reasonably synchronized----each philosopher will end up with one fork (left)--- each philosopher will block on trying to get the other fork (right)---nobody gets to eat at all

Attempt 2:

```
repeat forever:
    repeat:
        try to grab left fork
        try to grab right fork
        if both forks grabbed then break
        else put any grabbed forks back and take a short nap
    eat
    put forks back
    think
```

Attempt 3: same as before, but there is one pink hat--- only a philosopher with a pink hat will eat-- after eating, the philosopher passes the hat to another philosopher

yes it would, but…

- only one philosopher is eating at any given time
- but with 5 forks, 2 philosophers could be eating at the same time
- non-optimal use of resources, resulting in reduced parallelism
- called arbitrator solution

Attempt 4:

```
repeat forever:
    repeat:
        try to grab left fork
        try to grab right fork
        if both forks grabbed then break out of loop
        else put any grabbed forks back and take a short RANDOM nap
    eat
    put forks back
    think
```

random timeout mechanism for preventing deadlocks

- likely to work, often used in real world, for example in networking
- but… still a small chance for starvation:
  - if sleep is the same for all philosophers, no one gets to eat
  - in some cases some philosophers might never get to eat --or some philosophers will get to eat less often than others (fairness problem)

Attempt 5:

number the forks 1, 2, 3, 4, 5, each philosopher:

- picks up the fork with the smallest number first, then the larger number second
- if unable to pick up both forks, put a claimed fork down and take a tiny nap

this is a **resource hierarchy** solution – by establishing a **partial order** on resources

- starvation is still possible
- also not practical for large number of resources

dining philosopher problem.. many different solutions, not enough resources and more processes -> allocate resources -> might result in deadlock, livelock (stuff is being done but not making progress), starvation (one or more philosophers never get to eat), race conditions (if we have a bug in the implementation might corrupt resources -> not part of the problem but solution). All these are part of the solution, not the problem (due to design flaw)

Naive algorithm implementation

- implementation of even a naive solution presents problems
- consider algorithm for philosopher 'i':

```
// global variable representing chopstick state
// 0 = unavailable, 1 = available
int chopsticks[5];

while (true)
{
  sleep (s);  // think for s seconds
  while (!chopstick[i] || !chopstick[i+1]); // i+1 modulo 5 arithmetic
  chopstick[i] = false;
  chopstick[i+1] = false;
  sleep (m);   // eat for m seconds
  chopstick[i] = true;
  chopstick[i+1] = true;
}
```

Race condition

Philosopher i = 2:

```
while (true){
  sleep (n);  // thinking
  while (!chopstick[i]
        || !chopstick[i+1]);
  chopstick[i] = false;
  chopstick[i+1] = false;
  sleep (m);   // eating
  chopstick[i] = true;
  chopstick[i+1] = true;
}
```

Philosopher i = 3:

```
while (true){
  sleep (n);  // thinking
  while (!chopstick[i]
        || !chopstick[i+1]);
  chopstick[i] = false;
  chopstick[i+1] = false;
  sleep (m);   // eating
  chopstick[i] = true;
  chopstick[i+1] = true;
}
```

Race condition

```
load R1, [chopstick+3]
cmp  R1 1
```

```
load R1, [chopstick+3]
cmp  R1 1
```

Philosopher i = 2:

```
while (true){
  sleep (n);  // thinking
  while (!chopstick[i]
        || !chopstick[i+1]);
  chopstick[i] = false;
  chopstick[i+1] = false;
  sleep (m);   // eating
  chopstick[i] = true;
  chopstick[i+1] = true;
}
```

Philosopher i = 3:

```
while (true){
  sleep (n);  // thinking
  while (!chopstick[i]
        || !chopstick[i+1]);
  chopstick[i] = false;
  chopstick[i+1] = false;
  sleep (m);   // eating
  chopstick[i] = true;
  chopstick[i+1] = true;
}
```

Consider the following sequence:

```
Thread 1 : load R1, [chopstick+3]  [R1 = 0]
Thread 2 : load R1, [chopstick+3]  [R1 = 0]
Thread 1 : cmp  R1 1
Thread 2 : cmp  R1 1
```

Philosopher i = 2:

```
while (true){
  sleep (n);  // thinking
  while (!chopstick[i]
        || !chopstick[i+1]);
  chopstick[i] = false;
  chopstick[i+1] = false;
  sleep (m);   // eating
  chopstick[i] = true;
  chopstick[i+1] = true;
}
```

Philosopher i = 3:

```
while (true){
  sleep (n);  // thinking
  while (!chopstick[i]
        || !chopstick[i+1]);
  chopstick[i] = false;
  chopstick[i+1] = false;
  sleep (m);   // eating
  chopstick[i] = true;
  chopstick[i+1] = true;
}
```

- depending on the execution order, both philosophers could start eating at the same time
- they both would be inside their critical region

Philosopher with mutex

```
pthread_mutex_t mutex; // a lock, initialized elsewhere

while (true) {
  sleep (s);  // think
  pthread_mutex_lock(&mutex);
  while (!chopstick[i] || !chopstick[i+1]);
  chopstick[i] = false;
  chopstick[i+1] = false;
  pthread_mutex_unlock(&mutex);
  sleep (m);   // eating
  pthread_mutex_lock(&mutex);
  chopstick[i] = true;
  chopstick[i+1] = true;
  pthread_mutex_unlock(&mutex);
}
```

thread 2

thread 1

Deadlock !!!

- **critical section** - part of the program where a shared resource is accessed & may cause trouble
- **mutual exclusion** - ensuring only one process accesses a resource at a time, eg. only one process can enter critical section at any given time
- **mutex/lock** - mechanism to achieve mutual exclusion, two states, queue
- **deadlock** - a state where each process/thread is waiting on another to release a lock → no progress is made
- **livelock** - states of the processes change, but none are progressing
- **starvation** - one process does not get to run at all
- **unfairness** - not all processes get equal opportunity to progress

- **particular API in pthread library**

API	Description
`phread_mutex_init()`	create a mutex
`phread_mutex_destroy()`	destroy a mutex
`phread_mutex_lock()`	lock a mutex, block if already locked
`phread_mutex_trylock()`	lock a mutex, or fail (non-blocking version)
`phread_mutex_unlock()`	unlock a mutex

- pthread_create(thread, attr, start_routine, arg)
  - starts a thread, similar for fork()
- pthread_exit(status)
  - terminates the current thread, similar to exit()
- pthread_join(thread, status)
  - blocks the calling thread until the specified thread terminates, similar to wait()
- pthread_attr_init(attr) and pthread_attr_destroy(attr)
  - initializes / destroys thread attributes
- pthread_atfork()
  - used to register a callback in case fork() is called in a multithreaded program
- pthread_kill(threadid, signal)
  - one thread manually kills a target thread
- pthread_mutex_t mutex
  - initialize a mutex
- pthread_mutex_init(&mutex)
  - create a mutex
- pthread_mutex_destroy(&mutex)
  - destroy a mutex

- pthread_mutex_lock(&mutex)
  - lock a mutex, block if already locked
- pthread_mutex_trylock(&mutex)
  - lock a mutex, or fail (non-blocking version)
- pthread_mutex_unlock(&mutex)
  - unlock a mutex
- pthread_cont_t cond
  - initialize a condition variable
- pthread_cond_init(&cond, &attr)
  - create condition variable
- pthread_cond_destroy(&cond)
  - destroy a condition variable
- pthread_cond_broadcast(&cond)
  - wakes up all threads waiting on condition
- pthread_cond_wait(&cond, &mutex)
  - atomically releases mutex and causes the calling thread to block, until some other thread calls pthread_cond_signal(&cond)
- pthread_cond_signal(&cond)
  - wakes up one thread waiting on cond. If no thread is waiting on cond, the signal is lost
  - must be followed by pthread_mutex_unlock() if the blocked thread uses the same mutex
- pthread_spin_t lock
  - initialize a spinlock
- pthread_spin_init(&lock, int pshared)
  - initialize a spinlock
- pthread_spin_destroy(&lock)
  - destroy a spinlock
- pthread_spin_lock(&lock)
  - lock a spinlock, block if already locked
- pthread_spin_trylock(&lock)
  - lock a spinlock, or fail
- pthread_spin_unlock(&lock)
  - unlock a spinlock
- pthread_rwlock_*()

- pthread share-exclusive locks (also called reader-writer lock)
- Resource supports multiple concurrent readers, but only a single writer

- **Job scheduling algorithm (3 Q)**

First Come First Serve - No Starvation
Round Robin - No Starvation
Shortest Job First - Starvation
Shortest Remaining Time Next - Starvation
Shortest Process Next - Depends on initial prediction
Fair-Share Scheduling - Depends on scheduling algorithm used for each user
Priority Scheduling – Depends

First Come First Serve - No Starvation, Non-preemptive, LEAST context switches
CPU assigned in the order the processes request it, using a FIFO ready queue, new jobs are appended to the ready queue, a running job keeps the CPU until it is either finished, or it blocks.
Disadvantage: CPU-bound process will tie up the CPU, making I/O bound processes run for much longer (convoy effect)

Round Robin - No Starvation, Preemptive
Each process is assigned a time interval, called quantum. If the process exceeds the quantum, the process is preempted, and CPU is given to the next process in the ready queue, while the preempted process goes to back of ready queue.
- very small **Q** implies heavy overhead, but highly responsive system
- very large **Q** implies minimum overhead, but a non-responsive system
- performance of RR depends significantly on the size of the time quantum (**Q**) and the time required for a context switch (**S**)
- **Q** should be large compared to **S**

Shortest Job First - Starvation, Non-preemptive
Sorted on static execution time, based on submitted estimate. Execution time can be dynamically computed based on the history of CPU bursts.
Advantage: Minimum number of context switches
Optimal turnaround time if all jobs arrive simultaneously, minimize average waiting time
Disadvantage: Require advance knowledge of how long a job will execute, has potential for starvation

Shortest Remaining Time Next - Starvation, Preemptive
Next job picked based on remaining time = total time - time already spent on CPU
Ready queue is sorted based on remaining time, remaining time can be static, or dynamically calculated
similar advantages to SJF:

- **optimal turnaround time** even if jobs don't arrive at the same time
- but slightly more context switches

similar disadvantages to SJF:

- requires advance knowledge of how long a job will execute
- has a potential for job starvation
- and also needs to consider cost of context switch (More than N context switches for N processes)

Multi-level queue:

preemptive time-sharing scheduling algorithm that supports process priorities-- ready queue is partitioned into separate queues-- each queue can have a different scheduling algorithm-- a process is permanently assigned to one of the queues scheduling is done based on queues
Multi-level feedback queue:
similar to multilevel queues; but a process **can move** between queues (up or down in priority);
scheduler defined by

- number of queues
- scheduling algorithms for each queue, including different time slices
- when to move a process between queues *

- which queue a process will be assigned when the process needs service

Shortest Process Next - Preemptive
Ready queue is sorted by a predicted next CPU burst, calculated using exponential smoothing:

P = a*B + (1-a)*P'
P = new prediction, initialized to some value (e.g., 0)
P' = previous prediction
a = smoothing factor, commonly ½

Operation Environments	Scheduling Algorithms
Batch systems	• First come, first serve • Shortest job first • Shortest remaining time next
Interactive systems	• Round robin • Shortest process next • Fair share • Lottery • Multilevel queue • Multilevel feedback queue
Real-time systems	• Rate-monotonic scheduling • Earliest-deadline-first

Fair-Share Scheduling -
Fair-share based on owners (users) of the processes. Allocates the CPU among users/groups, instead of processes, all processes belonging to an owner have to share the owner's CPU share.
Priority Scheduling - Non pre-emptive
Lottery Scheduling - Each process gets N lottery tickets based on priority, if ticket picked, process gets time slice. No starvation
Multilevel Queues - ready queue partitioned into separate queues, scheduling depends on queue. Assignment is permanent.
Multilevel Feedback Queues - Like multilevel, but assignment may change . Minimizes starvation, reduces convoy effect

- **deadlock, unsafe state, safe state**

Deadlock ⇒ Unsafe State
Unsafe State !⇒ Deadlock
a system is in a **safe state** if there exists a sequence <P1, P2, …, Pn> of all running processes in the system where they can all finish, while allowing them to claim their maximum resources
a system is in an **unsafe state** if there does not exist such an execution sequence

- when a process requests an available resource, the system determines if granting such request would lead to a safe state --- if new state is safe, request is granted --if new state is not safe, request is denied and process waits

- **cycle + graph + deadlock (for single instance per resource type- use resource-allocation graph) (for multiple instances per resource type- banker's algorithm)**

No cycle ⇒ No deadlock
If cycle:

- one instance per resource type ⇒ guarantee deadlock
- multiple instance per resource type ⇒ possible deadlock

$P=\{P_1,\ldots,P_n\}=¿$ set of all PROCESSES in the system. use BOX
$R=\{R_1,\ldots,R_m\}=¿$ set of all RESOURCE TYPES in the system. use OVAL
$P_i \rightarrow R_j=¿$ REQUEST EDGE
$R_i \rightarrow P_j=¿$ ASSIGNMENT EDGE

- NO CYCLE ⇒ NO DEADLOCK
- CYCLE:
  - ONE INSTANCE PER RESOURCE TYPE ⇒ DEADLOCK

- OTHERWISE ⇒ CHECK MANUALLY

- **strict alternation (2 threads & running back and forth)**

Two processes, always run the first one for a little bit and then switch to the next one then switch back. One process has to wait for the other one to finish using resources to run.

Drawback: fast process has to wait for the slow one

Strict Alternation - A synchronization approach where two processes alternate entering their critical regions. Uses a global variable `turn = 0`

Drawbacks:

- Busy waiting
- Only works for 2 processes
- No progress - faster process is blocked by slower process

- **Neccesary condition for deadlock**

mutual exclusion - the involved resources are unshareable
hold and wait - a process holding at least one resource is waiting for more resources
no preemption - a resource can be released only by the process holding it (voluntary)
circular wait - there is a cycle

- **Difference deadlock avoicdance, recovery, protection**

- Ensure that the system will never enter a deadlock state (unsafe state) : deadlock **prevention**--- deadlock **avoidance**
- Allow the system to enter a deadlock state and then recover: deadlock **detection** ---- **recovery** from deadlock

Deadlock prevention is what we use to attack the necessary conditions

Deadlock prevention: any technique that attacks one of the 4 necessary conditions.

Avoid mutual exclusion

- mutual exclusion not required for shareable resources (read-only)
- spooling can help for some resource types

Avoid hold and wait condition

- whenever a process requests a resource, it does not hold any other resources

option 1: process must request all needed resources at beginning [lockn(mutex1, mutex2)

- or release resources before acquiring: unlockAndLock

option 2: process can request resources only when it has no resources

Avoid no preemption condition: if a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released, and process is suspended

- preempted resources are added to the list of resources for which the process is waiting
- process will be resumed when it can regain its old & new resources
- only works with resources for which we can save/restore the state (eg. CPU registers)
- complicated mechanism, possible starvation, non-optimal use of resources

```
Thread 1:

unlockAndLock( mutex2, mutex1);
// critical section
unlock( mutex2 );
unlock( mutex1 );
```

Avoid circular wait: most practical condition to avoid --- impose a total ordering on all resources, and require that each process requests resources in an increasing order of enumeration
– lock mutexes in the same order by all threads

Deadlock Example with Lock Ordering

```
void transaction(Account a1, Account a2, double amount) {
   mutex m1, m2;
   m1 = get_mutex(a1); // get exclusive access to account a1
   m2 = get_mutex(a2); // get exclusive access to account a2
   lock(m1);
      lock(m2);
         withdraw(a1, amount);
         deposit(a2, amount);
      unlock(m2);
   unlock(m1);
}
```

Imagine 2 transactions execute concurrently:

first transaction transfers from account A to account B - `transaction("A","B",20)`

second transaction transfers from account B to account A - `transaction("B","A",10)`

⇒ Possible Deadlock.

Deadlock Example with Lock Ordering

```
void transaction(Account a1, Account a2, double amount) {
   mutex m1, m2;
   m1 = get_mutex(a1); // get exclusive access to account a1
   m2 = get_mutex(a2); // get exclusive access to account a2
   if( id(a1) > id(a2))
      swap( m1, m2);
   lock(m1);
      lock(m2);
         withdraw(a1, amount);
         deposit(a2, amount);
      lock(m2);
   lock(m1);
}
```

Alternatively:

```
if( id(a1) > id(a2)) {
    lock(m1); lock(m2);
} else {
    lock(m2); lock(m1);
}
```

Avoiding circular wait condition by ordering resources.

Deadlock Avoidance:

deadlock prevention schemes can lead to low resource utilization

- **deadlock avoidance** can increase resource utilization if some **a priori information** is available
- simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
- a deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

Deadlock detection: Detection algorithm--

** with single instance per resource type-- maintain an up-to-date **wait-for** graph

- nodes are processes
- edge Pi → Pj exists if Pi is waiting for Pj
- can be obtained by **collapsing** resource-allocation graph

- periodically invoke an algorithm that searches for a cycle in the graph
- if there is a cycle, there exists a deadlock

** with multiple instances per resource type:

similar to banker's algorithm

- we try to determine if there is a sequence in which running processes can **finish** executing
- but assumes best case scenario - a process that is given its requested resources will finish without asking for more resources, and then releases all its resources
- does not need Max[] or Need[] matrices

detection algorithms are expensive

- we cannot invoke them on every resource request

- other ideas for invoking detection:
  - check every few minutes
  - check when CPU goes idle

- when, and how often depends on:
    - how often a deadlock is likely to occur?
    - how many processes will be affected?
        - one for each disjoint cycle
- if we check too often - we spend too many CPU cycles on useless work
- if we don't check often enough - there may be many cycles in the resource graph and we would not be able to tell which of the many deadlocked processes “caused” the deadlock

Deadlock recovery: process termination--

<u>process rollback-</u>- programs can be implemented to periodically save their current state (**checkpoint**)

- when restarted, the program detects a checkpoint and resumes computation from last checkpoint (**rollback**)
- programs can checkpoint themselves just before requesting resources

- when deadlock is detected, a program can be terminated and re-scheduled to run later
    - eg. after the other affected deadlocked processes are done
- does not work well with all resource types (eg. printer)
- useful for long computations / simulation

<u>Resource preemption</u>

similar idea to rollback, but instead of checkpointing the program, we checkpoint the resources of the program

- only works with some resource types
- when deadlock occurs:
    - pick a victim process
    - suspend victim process
    - save state of victim's resources
    - give victim's resources to other deadlocked processes
    - when the other processes release the resources, restore resource states
    - return resources to the victim
    - unsuspend the victim

Banker- deadlock avoicdance; Spooling- mutual exclusion

- **Banker algorithm**

show if something is in safe state or not. whether a particular request can be granted or not

- Deadlock detection algorithm

n = number of processes

- m = number of resource types
- Available[] = vector of length m indicates the number of available resources of each type
- Allocation[][] = n x m matrix - current allocation of resources per process

Request[][] = n x m matrix - current requests for resources per process

```
1. Initialize:
   Work = Available
   Finish[i] = (Allocation[i] == 0) for all i = 1,2,3, … n
2. Find an index i such that:
      Finish[i] == false and Request[i] ≤ Work
   If no such i exists, go to step 4
3. Work = Work + Allocation[i]
   Finish[i] = true
   go to step 2
4. If Finish[i] == true for 1 ≤ i ≤ n, then system is not in deadlock
   Otherwise, all processes Pi for which Finish[i] == false are deadlocked
```

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state.

- Memory allocation algorithm

(first fit, best fit, next fit, worst fit)

Best Fit - Find the smallest hole that is big enough, leftover (tiny) space becomes new hole
First Fit - Find the first hole that is big enough, leftover space becomes new hole
Next Fit - Same as first fit, but starting at the location of last placement
Worst Fit - Find the largest hole, leftover space is likely to be usable
Quick Fit - Maintain separate lists for common request sizes... Faster search, more complicated management

- Memory fragmentation (internal and external)

- Page replacement algorithm (FIFO, LRU, Clock)
- Virtual memomory, logical address, physical addresses
- Copy on Write
- Page fault
- Thrashing (process thrashing/ system thrashing)
- file system (generic ones)
- file system (inode)
- differences (contiguous, FAT, inode) Why choose one versus the otherwise
- Disk I/O scheduling
- Semaphore
- **TLB Translation Lookaside Buffer** - A special fast-lookup hardware cache, extremely fast and extremely small (64 to 1K entries). On TLB miss, value is loaded into TLB for faster access next time.

TLB is often implemented as associative memory: hardware capable of fast parallel search based on content. Given a page number, TLB will return the corresponding frame number in constant amount of time.

Short answer: (2 marks/ question)

- Calculate from virtual memory
- Logical address to physical addresses
- Page table (number of page table entries)
- thrashing
- Memory allocation
- Page table entries (what do they contain? With the bit)
- **Generic schedule (preemptive, non-preemptive)**

non-preemptive — context switch happens only voluntarily

- multitasking is possible, but only through cooperation
- process runs until it does a blocking syscall (eg. I/O), terminates, or voluntarily yields CPU
- example: FCFS

preemptive — context switch can happen without cooperation

- usually as a direct or indirect result of an interrupt, but not limited to clock interrupt

  eg. new job is added, existing process is unblocked
- example: SRTN

preemptive time-sharing — special case of preemptive

- processes are context switched periodically, usually to enforce time-slice policy
- implemented through clock interrupts
- without a clock, only cooperative multitasking (non-preemptive) is possible
- example: RR

response time: how long before you get first feedback, often response = start – arrival

turnaround time: time from arrival to finish, turnaround = finish – arrival

CPU time: how much time the process spent on CPU

waiting time: total time spent it waiting queue, waiting = turnaround - CPU - I/O

- Generic I/O
- Fork
- Premption and context switch
- Low level atomic (swap, test & set, CAS)
- swapping question (different type of swap)
- bash
- FAT (file allocation)
- Paging question, paging algorithm
- Effective access time handling EAT

Long answer (8 marks/ question)

- File allocation
- Page replacement (Reference string, the algorithm to use LRU, Opt, Clock), how many page fault
- Page table question (calcualte about virtual memory)
- Convert from physical addresses to logical addresses and vice versa
- I/O scheduling question (the algorithm)
- Inode question (indirect pointer) – calcualte to maximize the file size supported
- Deadlock (single instance per resource type)
- Calculate safe state (banker algorithm)
- GANTT chart (for the schedule)
- Multi-thread code (what is possible outcome, race condition, what is the possible out come)

BANKER ALGORITHM: Banker's algorithm tries to find whether there is an execution sequence that would finish all running processes, assuming worst case scenario - that each process requests its maximum declared allocation as the next step.

- another avoidance algorithm
- more general than resource-allocation graph algorithm
- works with multiple instances per resource type
- requirements:
  - each process must declare maximum resources at the beginning
  - when a process requests a resource it may have to wait (even if resource is available)
  - when a process gets all its resources it must return them in a finite amount of time

Let `n` = number of processes, and `m` = number of resources types.

- **Available** - vector of length *m*
  - `Available[j]=k` means there are *k* instances of resource type R_j available
- **Max** - matrix *n x m*
  - `Max[i,j]=k` means that process P_i may request at most *k* instances of resource type R_j
  - `Max[i]` represents the i-th row of Max
- **Allocation** - matrix *n x m*
  - `Allocation[i,j]=k` means that process P_i is currently allocated *k* instances of R_j
- **Need** - matrix *n x m*
  - `Need[i,j]=k` means that P_i may need *k* more instances of R_j to complete its task
- Note: `Need[i,j] = Max[i,j] – Allocation[i,j]`
  - only 2 of Need/Max/Allocation are needed, the 3rd can be computed on the fly

```
1.  Initialize temporary vectors:
    Work = Available ; vector copy
    Finish[i] = false for i = 0, 1, …, n-1
2.  Find an i such that
      Finish[i] = false AND Need[i] ≤ Work ; vector comparison
    If no such i exists, go to step 4
3.  Update Work and Finish:
      Work = Work + Allocation[i] ; vector sum
      Finish[i] = true
    go to step 2
4.  If Finish[i] == true for all i, return true ; system is in a safe state
    else return false ; system is not in a safe state
```

Complexity:

O(n*n*m)

Algorithm to determine if request can be granted for process P_i

Inputs: (Request, i)

- Request is a vector - where Request[j] = k means process P_i wants k instances of R_j
- process number i

Algorithm:

- If Request[] > Need[i] ⇒ raise error, process has exceeded its maximum claim
- If Request[] > Available ⇒ then P_i must wait, since resources are not available
- Pretend to allocate Request[], and see if the state is still safe:
  - Save current state of Available, Allocation and Need
  - Available = Available - Request
  - Allocation[i] = Allocation[i] + Request
  - Need[i] = Need[i] - Request
- check if state still safe using Banker's algorithm
- If state safe ⇒ the request can be granted (resources are allocated to P_i), discard the saved state
- If unsafe ⇒ request must block, P_i must wait, restore the saved state

1KB= 2^{10} Byte;
1MB= 2^{20} Bytes;
1 GB= 2^{30} Bytes;
1TB= 2^{40} Bytes