Discussion 5

Starvation

10/16/24

Staff

Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	Project 1 Due	Project 2 Release				
					Project 2 Design Doc Due	
Homework 3 Due	Homework 4 Release					

Starvation

Starvation

Important to prevent **starvation**, a situation where a thread fails to make progress for an indefinite period of time.

- Scheduling policy never runs a particular thread on the CPU.
- Threads wait for each other or are spinning in a way that will never be resolved.

Strict Priority

Schedules task with highest priority.

- Tasks with same priority can be scheduled in some other fashion (e.g. RR, FCFS)

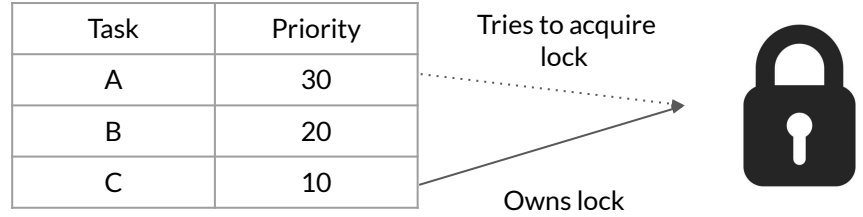
Ensures important tasks get to run first.

Can suffer from starvation for lower priority threads.

Priority inversion can occur where a higher priority task is blocked waiting on a lower priority task.

- Medium priority task (in between higher and lower) will run (i.e. medium priority starves higher priority).
- Fix with **priority donation** where lower priority task is *temporarily* granted same priority as higher task.
    - Gives lower priority task the same **effective priority** as the higher priority task.
    - Once lower priority task is no longer blocking the higher priority task, the lower priority task returns to its **base priority**.

Task	Priority
A	30
B	20
C	10



Lottery

Gives each task some number of lottery tickets

- At each time slice a random ticket is drawn. The task holding that ticket is granted the resource.
- On expectation, each task uses the resource for a time proportional to the number of tickets it holds.

Assign ticket numbers in a variety of schemes.

- Approximate SRTF by giving more tickets to shorter jobs and fewer tickets to longer jobs (i.e. use tickets as a measure of priority).
- Make sure every job gets at least one ticket to avoid starvation.

Stride

Deterministic version of a lottery scheduler.

- Each task given some number of tickets n_i.
- **Stride** is a number inversely proportional to the number of tickets, typically calculated as W/n_i where W is a large number.

On every time slice, task with the lowest **pass** is chosen.

- Pass is initialized to min(existing tasks' pass values) when the task starts.
- When a task is chosen, pass += stride.
- Smaller stride → task runs more often.

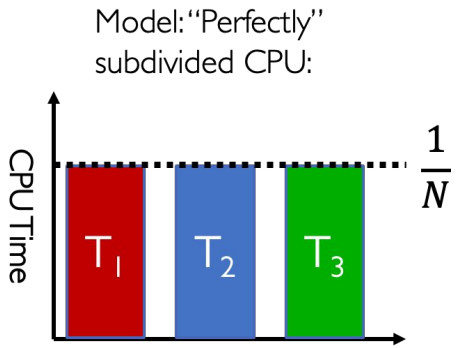
Task	Tickets	Stride (W = 10000)
A	100	100
B	50	200
C	250	40

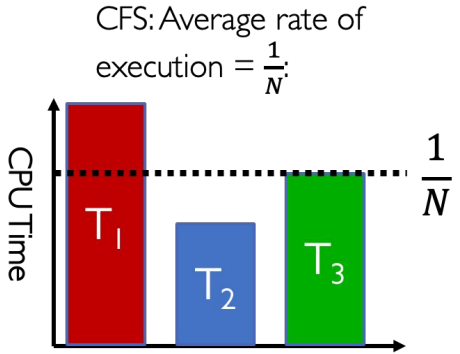
Pass		
A	B	C
0	0	0
100	0	0
100	200	0
100	200	40
100	200	80
100	200	120
200	200	120
200	200	160

Linux Completely Fair Scheduler (CFS)

Aims to give each task an equal share of the CPU.

- Gives illusion that each task executes simultaneously on 1/n of the CPU.
- Hardware limitations of giving out CPU in full time slices → scheduler tracks CPU time per task and schedules task to match *average* rate of execution.
- When choosing a task to run, choose the one with minimum CPU time.
    - Efficiently do this using a heap like structure (logarithmic with respect to number of tasks).





Earliest Eligible Virtual Deadline First (EEVDF)

EEVDF Scheduler

English

The "Earliest Eligible Virtual Deadline First" (EEVDF) was first introduced in a scientific publication in 1995 [1]. The Linux kernel began transitioning to EEVDF in version 6.6 (as a new option in 2024), moving away from the earlier Completely Fair Scheduler (CFS) in favor of a version of EEVDF proposed by Peter Zijlstra in 2023 [2-4]. More information regarding CFS can be found in CFS Scheduler.

https://www.kernel.org/doc/html/next/scheduler/sched-eevdf.html

A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems

Ion Stoica * Hussein Abdel-Wahab † Kevin Jeffay‡ Sanjoy K. Baruah §
Johannes E. Gehrke ¶ C. Greg Plaxton ||

Abstract

We propose and analyze a proportional share resource allocation algorithm for realizing real-time performance in time-shared operating systems. Processes are assigned a weight which determines a share (percentage) of the resource they are to receive. The resource is then allocated in discrete-sized time quanta in such a manner that each process makes progress at a precise, uniform rate. Proportional share allocation algorithms are of interest because (1) they provide a time quantum. In addition, the algorithm provides support for dynamic operations, such as processes joining or leaving the competition, and for both fractional and non-uniform time quanta. As a proof of concept we have implemented a prototype of a CPU scheduler under FreeBSD. The experimental results shows that our implementation performs within the theoretical bounds and hence supports real-time execution in a general purpose operating system.

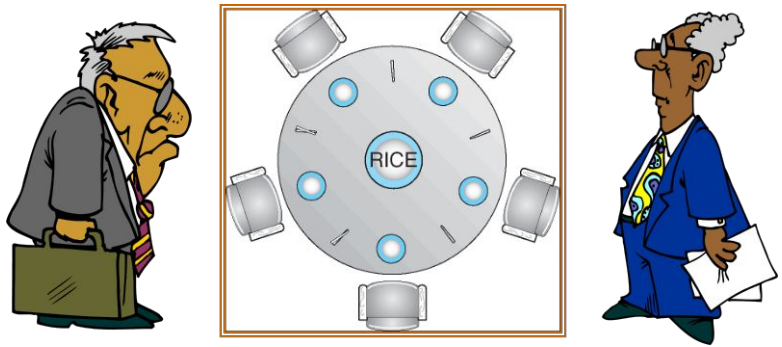
Deadlock

Deadlock is a situation where there is a cycle of waiting among a set of threads, where each thread waits for some other thread in the cycle to take some action.

- Special form of starvation (stronger condition).

Necessary but *not sufficient* conditions for deadlock.

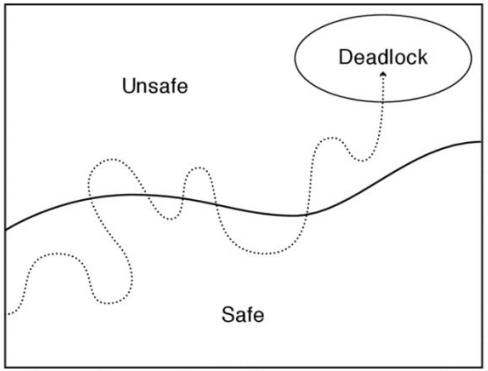
- **Mutual exclusion and bounded resources**: finite number of threads (usually one) can simultaneously use a resource.
- **Hold and wait**: thread holds one resource while waiting to acquire additional resources held by other threads.
- **No preemption**: once a thread acquires a resource, its ownership cannot be revoked until the thread acts to release it.
- **Circular waiting**: there exists a set of waiting threads such that each thread is waiting for a resource held by another.



Deadlock

Four main ways to handle deadlock.

- **Denial**: pretend deadlock is not a problem (i.e. ostrich algorithm).
- **Prevention**: write systems that don't result in deadlock.
    - Prevent one of the necessary conditions for deadlock.
- **Recovery**: let deadlock happen and recover from it afterwards.
    - Terminate a thread, forcing it to give up resources.
    - Roll back actions (danger of deadlocking in the same way)
- **Avoidance**: dynamically delay resource requests, so deadlock doesn't happen.
    - Check if a resource would result in a deadlock when a thread requests a resource?
        - Too late since thread might end up in a **unsafe state** where there isn't a deadlock yet but there is potential for a pattern of resource requests that unavoidably leads to deadlock.
    - System must always remain in a **safe state** where the system can delay resource requests to prevent deadlock.
    - Check for **unsafe state** (not deadlock) on a request.



Deadlock

Banker's algorithm.

Require:

available, array of how much of each resource is available

alloc, 2D array where the i-th element is how many of each resource thread i currently holds.

max, 2D array where the i-th element is the max number of resources resource thread i is requesting.

```
unfinished ← all threads
done ← false
while done is false do
    done ← true
    for thread in unfinished do
        if max[thread] - alloc[thread] ≤ available then
            remove thread from unfinished
            available ← available + alloc[thread]
            done ← false
        end if
    end for
end while
```

Deadlock

Banker's algorithm.

Summary:

- Takes a snapshot of program execution state: which resources are available, owned, or being requested
- If no threads remain in the unfinished queue:
  - there exists an execution of threads {T1, T2, …, Tn}. our program is in a **SAFE state**.
- Else:
  - Our program is in an **UNSAFE state**. This means we can't guarantee that deadlock won't occur, even if it hasn't occurred yet.
- What does the Banker's algorithm do?
  - If the program is in an unsafe state, it will deny the request for resources.
  - It is meant to be a proactive check — part of **deadlock avoidance**

Deadlock

Total	
R1	R2
4	5

Available = Total - Allocated	
R1	R2
1	1

Is this a safe state?

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

Allocated		
	R1	R2
T1	3	2
T2	0	2

Request		
	R1	R2
T1	3	5
T2	1	2

Needed = Request - Allocated		
	R1	R2
T1	0	3
T2	1	0

Deadlock

Total	
R1	R2
4	5

Available = Total - Allocated	
R1	R2
1	1

Is this a safe state?

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

Schedule T2

Allocated		
	R1	R2
T1	3	2
T2	0	2

Request		
	R1	R2
T1	3	5
T2	1	2

Needed = Request - Allocated		
	R1	R2
T1	0	3
T2	1	0

Deadlock

Total	
R1	R2
4	5

Available = Total - Allocated	
R1	R2
0	1

Is this a safe state?

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

T2 is allocated all the resources it needs to run

Allocated		
	R1	R2
T1	3	2
T2	1	2

Request		
	R1	R2
T1	3	5
T2	1	2

Needed = Request - Allocated		
	R1	R2
T1	0	3
T2	0	0

Deadlock

Total	
R1	R2
4	5

Available = Total - Allocated	
R1	R2
1	3

Is this a safe state?

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

Release T2's resources when it finishes

Allocated		
	R1	R2
T1	3	2
T2	0	0

Request		
	R1	R2
T1	3	5
T2	1	2

Needed = Request - Allocated		
	R1	R2
T1	0	3
T2	0	0

Deadlock

Total	
R1	R2
4	5

Available = Total - Allocated	
R1	R2
1	3

Is this a safe state?

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

Schedule T1

Allocated		
	R1	R2
T1	3	2
T2	0	0

Request		
	R1	R2
T1	3	5
T2	1	2

Needed = Request - Allocated		
	R1	R2
T1	0	3
T2	0	0

Deadlock

Total	
R1	R2
4	5

Available = Total - Allocated	
R1	R2
1	0

Is this a safe state?

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

T1 is allocated all the resources it needs to run

Allocated		
	R1	R2
T1	3	5
T2	0	0

Request		
	R1	R2
T1	3	5
T2	1	2

Needed = Request - Allocated		
	R1	R2
T1	0	0
T2	0	0

Deadlock

Total	
R1	R2
4	5

Available = Total - Allocated	
R1	R2
4	5

Is this a safe state?

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

Release T1's resources when it finishes

Allocated		
	R1	R2
T1	0	0
T2	0	0

Request		
	R1	R2
T1	3	5
T2	1	2

Needed = Request - Allocated		
	R1	R2
T1	0	0
T2	0	0

Deadlock

Total	
R1	R2
4	5

Available = Total - Allocated	
R1	R2
1	1

Is this a safe state?

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

What if T2 needed 2 amounts of R1 instead of just 1?

Allocated		
	R1	R2
T1	3	2
T2	0	2

Request		
	R1	R2
T1	3	5
T2	**2**	2

Needed = Request - Allocated		
	R1	R2
T1	0	3
T2	**2**	0

Deadlock

Total	
R1	R2
4	5

Available = Total - Allocated	
R1	R2
1	1

Is this a safe state?

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

What if T2 needed 2 amounts of R1 instead of just 1?

System is in an unsafe state!

Allocated		
	R1	R2
T1	3	2
T2	0	2

Request		
	R1	R2
T1	3	5
T2	2	2

Needed = Request - Allocated		
	R1	R2
T1	0	3
T2	2	0

Concept Check

1. In what sense is Linux CFS completely fair?

2. How can you easily implement lottery scheduling?

3. Is stride scheduling prone to starvation?

4. When using stride scheduling, if a task is more urgent, should it be assigned a larger stride or a smaller stride?

Concept Check

1. In what sense is Linux CFS completely fair?
   Give all tasks equal access to the CPU.

2. How can you easily implement lottery scheduling?

3. Is stride scheduling prone to starvation?

4. When using stride scheduling, if a task is more urgent, should it be assigned a larger stride or a smaller stride?

Concept Check

1. In what sense is Linux CFS completely fair?

   Give all tasks equal access to the CPU.

2. How can you easily implement lottery scheduling?

   Select a random number from 1 to N, where N is the total number of tickets.

3. Is stride scheduling prone to starvation?

4. When using stride scheduling, if a task is more urgent, should it be assigned a larger stride or a smaller stride?

Concept Check

1. In what sense is Linux CFS completely fair?

   Give all tasks equal access to the CPU.

2. How can you easily implement lottery scheduling?

   Select a random number from 1 to N, where N is the total number of tickets.

3. Is stride scheduling prone to starvation?

   No. All threads will deterministically run.

4. When using stride scheduling, if a task is more urgent, should it be assigned a larger stride or a smaller stride?

Concept Check

1. In what sense is Linux CFS completely fair?

   Give all tasks equal access to the CPU.

2. How can you easily implement lottery scheduling?

   Select a random number from 1 to N, where N is the total number of tickets.

3. Is stride scheduling prone to starvation?

   No. All threads will deterministically run.

4. When using stride scheduling, if a task is more urgent, should it be assigned a larger stride or a smaller stride?

   Smaller stride.

Simple Priority Scheduler

Let's implement a new scheduler in Pintos called the simple priority scheduler (SPS). We will just split threads into two priorities: high (1) and low (0). High priority threads should always be scheduled before low priority threads. Turns out we can do this without expensive list operations.

```
struct thread {
  ...
  int priority;
  struct list_elem elem;
  ...
}

struct list ready_list;

void thread_unblock (struct thread *t)  {
  ASSERT(is_thread(t));
  enum intr_level old_level;
  old_level = intr_disable();
  ASSERT(t->status == THREAD_BLOCKED);

  if (__________) {
    ____________________;
  } else {
    ____________________;
  }
  t->status = THREAD_READY;
  intr_set_level(old_level);
}
```

1. Complete the blanks of `thread_unblock` to implement SPS.

Simple Priority Scheduler

Let's implement a new scheduler in Pintos called the simple priority scheduler (SPS). We will just split threads into two priorities: high (1) and low (0). High priority threads should always be scheduled before low priority threads. Turns out we can do this without expensive list operations.

```
struct thread {
  ...
  int priority;
  struct list_elem elem;
  ...
}

struct list ready_list;

void thread_unblock (struct thread *t)  {
  ASSERT(is_thread(t));
  enum intr_level old_level;
  old_level = intr_disable();
  ASSERT(t->status == THREAD_BLOCKED);

  if (t->priority == 1) {
    list_push_front(&ready_list, &t->elem);
  } else {
    list_push_back(&ready_list, &t->elem);
  }
  t->status = THREAD_READY;
  intr_set_level(old_level);
}
```

1. Complete the blanks of `thread_unblock` to implement SPS.

Simple Priority Scheduler

Let's implement a new scheduler in Pintos called the simple priority scheduler (SPS). We will just split threads into two priorities: high (1) and low (0). High priority threads should always be scheduled before low priority threads. Turns out we can do this without expensive list operations.

```
struct thread {
  ...
  int priority;
  struct list_elem elem;
  ...
}

struct list ready_list;

void thread_unblock (struct thread *t)  {
  ASSERT(is_thread(t));
  enum intr_level old_level;
  old_level = intr_disable();
  ASSERT(t->status == THREAD_BLOCKED);

  if (t->priority == 1) {
    list_push_front(&ready_list, &t->elem);
  } else {
    list_push_back(&ready_list, &t->elem);
  }
  t->status = THREAD_READY;
  intr_set_level(old_level);
}
```

2. In order for this scheduler to be "fair", briefly describe when you would make a thread high priority and when you would make a thread low priority.

3. If we let the user set priorities of this scheduler with set_priority, why might this scheduler be preferable to the normal Pintos priority scheduler?

4. How can we trade off between the coarse granularity of SPS and the super fine granularity of normal priority scheduling? Assume we still want a fast insert.

Simple Priority Scheduler

Let's implement a new scheduler in Pintos called the simple priority scheduler (SPS). We will just split threads into two priorities: high (1) and low (0). High priority threads should always be scheduled before low priority threads. Turns out we can do this without expensive list operations.

```
struct thread {
  ...
  int priority;
  struct list_elem elem;
  ...
}

struct list ready_list;

void thread_unblock (struct thread *t)  {
  ASSERT(is_thread(t));
  enum intr_level old_level;
  old_level = intr_disable();
  ASSERT(t->status == THREAD_BLOCKED);

  if (t->priority == 1) {
    list_push_front(&ready_list, &t->elem);
  } else {
    list_push_back(&ready_list, &t->elem);
  }
  t->status = THREAD_READY;
  intr_set_level(old_level);
}
```

2. In order for this scheduler to be "fair", briefly describe when you would make a thread high priority and when you would make a thread low priority.
   Downgrade priority when thread uses up its quanta, upgrade priority when it voluntarily yields/gets blocked.

3. If we let the user set priorities of this scheduler with set_priority, why might this scheduler be preferable to the normal Pintos priority scheduler?

4. How can we trade off between the coarse granularity of SPS and the super fine granularity of normal priority scheduling? Assume we still want a fast insert.

Simple Priority Scheduler

Let's implement a new scheduler in Pintos called the simple priority scheduler (SPS). We will just split threads into two priorities: high (1) and low (0). High priority threads should always be scheduled before low priority threads. Turns out we can do this without expensive list operations.

```
struct thread {
  ...
  int priority;
  struct list_elem elem;
  ...
}

struct list ready_list;

void thread_unblock (struct thread *t)  {
  ASSERT(is_thread(t));
  enum intr_level old_level;
  old_level = intr_disable();
  ASSERT(t->status == THREAD_BLOCKED);

  if (t->priority == 1) {
    list_push_front(&ready_list, &t->elem);
  } else {
    list_push_back(&ready_list, &t->elem);
  }
  t->status = THREAD_READY;
  intr_set_level(old_level);
}
```

2. In order for this scheduler to be "fair", briefly describe when you would make a thread high priority and when you would make a thread low priority.
   Downgrade priority when thread uses up its quanta, upgrade priority when it voluntarily yields/gets blocked.

3. If we let the user set priorities of this scheduler with set_priority, why might this scheduler be preferable to the normal Pintos priority scheduler?
   Insert operations are cheaper, good approximation.

4. How can we trade off between the coarse granularity of SPS and the super fine granularity of normal priority scheduling? Assume we still want a fast insert.
   Have a fixed number of priorities and a queue for each priority.

Simple Priority Scheduler

Let's implement a new scheduler in Pintos called the simple priority scheduler (SPS). We will just split threads into two priorities: high (1) and low (0). High priority threads should always be scheduled before low priority threads. Turns out we can do this without expensive list operations.

```
struct thread {
  ...
  int priority;
  struct list_elem elem;
  ...
}

struct list ready_list;

void thread_unblock (struct thread *t)  {
  ASSERT(is_thread(t));
  enum intr_level old_level;
  old_level = intr_disable();
  ASSERT(t->status == THREAD_BLOCKED);

  if (t->priority == 1) {
    list_push_front(&ready_list, &t->elem);
  } else {
    list_push_back(&ready_list, &t->elem);
  }
  t->status = THREAD_READY;
  intr_set_level(old_level);
}
```

2. In order for this scheduler to be "fair", briefly describe when you would make a thread high priority and when you would make a thread low priority.
   Downgrade priority when thread uses up its quanta, upgrade priority when it voluntarily yields/gets blocked.

3. If we let the user set priorities of this scheduler with set_priority, why might this scheduler be preferable to the normal Pintos priority scheduler?
   Insert operations are cheaper, good approximation to priority scheduling.

4. How can we trade off between the coarse granularity of SPS and the super fine granularity of normal priority scheduling? Assume we still want a fast insert.

Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows.

1. Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

	A	B	C
Total	7	8	9
Available			

	Current			Max			Needed		
	A	B	C	A	B	C	A	B	C
T1	0	2	2	4	3	3			
T2	2	2	1	3	6	9			
T3	3	0	4	3	1	5			
T4	1	3	1	3	3	4			

Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows.

	A	B	C
Total	7	8	9
Available	1	1	1

1. Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

   Fill out available and needed tables.

	Current			Max			Needed		
	A	B	C	A	B	C	A	B	C
T1	0	2	2	4	3	3	4	1	1
T2	2	2	1	3	6	9	1	4	8
T3	3	0	4	3	1	5	0	1	1
T4	1	3	1	3	3	4	2	0	3

Banker’s Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows.

1. Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

	A	B	C
Total	7	8	9
Available	1	1	1

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

Execution Order:

	Current			Max			Needed		
	A	B	C	A	B	C	A	B	C
T1	0	2	2	4	3	3	4	1	1
T2	2	2	1	3	6	9	1	4	8
T3	3	0	4	3	1	5	0	1	1
T4	1	3	1	3	3	4	2	0	3

Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows.

1. Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

	A	B	C
Total	7	8	9
Available	1	1	1

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

Execution Order:

	Current			Max			Needed		
	A	B	C	A	B	C	A	B	C
T1	0	2	2	4	3	3	4	1	1
T2	2	2	1	3	6	9	1	4	8
T3	3	0	4	3	1	5	0	1	1
T4	1	3	1	3	3	4	2	0	3

Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows.

	A	B	C
Total	7	8	9
Available	4	1	5

1. Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

Execution Order: T3

	Current			Max			Needed		
	A	B	C	A	B	C	A	B	C
T1	0	2	2	4	3	3	4	1	1
T2	2	2	1	3	6	9	1	4	8
T3	0	0	0	3	1	5	0	0	0
T4	1	3	1	3	3	4	2	0	3

Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows.

	A	B	C
Total	7	8	9
Available	4	1	5

1. Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

Execution Order: T3

	Current			Max			Needed		
	A	B	C	A	B	C	A	B	C
T1	0	2	2	4	3	3	4	1	1
T2	2	2	1	3	6	9	1	4	8
T3	0	0	0	3	1	5	0	0	0
T4	1	3	1	3	3	4	2	0	3

Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows.

	A	B	C
Total	7	8	9
Available	4	3	7

1. Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

Execution Order: T3, T1

	Current			Max			Needed		
	A	B	C	A	B	C	A	B	C
T1	0	0	0	4	3	3	0	0	0
T2	2	2	1	3	6	9	1	4	8
T3	0	0	0	3	1	5	0	0	0
T4	1	3	1	3	3	4	2	0	3

Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows.

1. Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

	A	B	C
Total	7	8	9
Available	4	3	7

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

Execution Order: T3, T1

	Current			Max			Needed		
	A	B	C	A	B	C	A	B	C
T1	0	0	0	4	3	3	0	0	0
T2	2	2	1	3	6	9	1	4	8
T3	0	0	0	3	1	5	0	0	0
T4	1	3	1	3	3	4	2	0	3

Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows.

	A	B	C
Total	7	8	9
Available	5	6	8

1. Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

Execution Order: T3, T1, T4

	Current			Max			Needed		
	A	B	C	A	B	C	A	B	C
T1	0	0	0	4	3	3	0	0	0
T2	2	2	1	3	6	9	1	4	8
T3	0	0	0	3	1	5	0	0	0
T4	0	0	0	3	3	4	0	0	0

Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows.

	A	B	C
Total	7	8	9
Available	5	6	8

1. Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

Execution Order: T3, T1, T4

	Current			Max			Needed		
	A	B	C	A	B	C	A	B	C
T1	0	0	0	4	3	3	0	0	0
T2	2	2	1	3	6	9	1	4	8
T3	0	0	0	3	1	5	0	0	0
T4	0	0	0	3	3	4	0	0	0

Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows.

1. Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

	A	B	C
Total	7	8	9
Available	7	8	9

1. Schedule threads where needed[thread] ≤ available.
2. Release resources when thread is finished.
   available += allocated[thread]
3. Repeat.

Execution Order: T3, T1, T4, T2

	Current			Max			Needed		
	A	B	C	A	B	C	A	B	C
T1	0	0	0	4	3	3	0	0	0
T2	0	0	0	3	6	9	0	0	0
T3	0	0	0	3	1	5	0	0	0
T4	0	0	0	3	3	4	0	0	0

Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows.

2. If the total number of C instances is 8 instead of 9, is the system still in a safe state?

	A	B	C
Total	7	8	8
Available			

	Current			Max			Needed		
	A	B	C	A	B	C	A	B	C
T1	0	2	2	4	3	3	4	1	1
T2	2	2	1	3	6	9	1	4	8
T3	3	0	4	3	1	5	0	1	1
T4	1	3	1	3	3	4	2	0	3

Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows.

2. If the total number of C instances is 8 instead of 9, is the system still in a safe state?

	A	B	C
Total	7	8	8
Available	1	1	0

	Current			Max			Needed		
	A	B	C	A	B	C	A	B	C
T1	0	2	2	4	3	3	4	1	1
T2	2	2	1	3	6	9	1	4	8
T3	3	0	4	3	1	5	0	1	1
T4	1	3	1	3	3	4	2	0	3

Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows.

2. If the total number of C instances is 8 instead of 9, is the system still in a safe state?
   Unsafe state since no thread is able to run.

	A	B	C
Total	7	8	8
Available	1	1	0

	Current			Max			Needed		
	A	B	C	A	B	C	A	B	C
T1	0	2	2	4	3	3	4	1	1
T2	2	2	1	3	6	9	1	4	8
T3	3	0	4	3	1	5	0	1	1
T4	1	3	1	3	3	4	2	0	3