Discussion 5: Starvation

October 16, 2024

Contents

1 Starvation

When designing scheduling policies, it's important to prevent **starvation**, a situation where a thread fails to make progress for an indefinite period of time. If a scheduling policy never runs a particular thread on the CPU, the thread will starve. Threads can also starve if they wait for each other or are spinning in a way that will never be resolved.

Strict Priority

A **strict priority scheduler** schedules the task with the highest priority. If multiple tasks have the same priority, they can be scheduled in some RR fashion. This ensures that important tasks get to run first but doesn't maintain fairness. Evidently, a strict priority scheduler suffers from starvation for lower priority threads.

Priority Inversion

With a priority scheduler (e.g. strict priority), **priority inversion** can become a problem where a higher priority task is blocked waiting on a lower priority task. As a result, a medium priority task (in between the higher and lower priority) will be run by the scheduler, resulting in the medium priority task starving the higher priority task. A common example of this is when a higher priority task tries to acquire a mutex that the lower priority task already holds.

To address priority inversion, the scheduler can use **priority donation** where the lower priority task is *temporarily* granted the same priority as the higher priority task, so it can run. In this scenario, the lower priority thread is said to have an **effective priority** of the higher priority task. Once the higher priority task is no longer blocked on the lower priority task, the scheduler would demote it back to its **base priority**.

Lottery

A **lottery scheduler** gives each task some number of lottery tickets. At each time slice, a random ticket is drawn, and the task holding that ticket is granted the resource. On expectation, the time each task is allocated the resource for will be proportional to the number of tickets it holds.

Ticket numbers can be assigned in a variety of schemes. For instance, the scheduler could approximate SRTF by giving more tickets to shorter jobs and less tickets to longer jobs. Essentially, the number of tickets serve as a measure of priority in some sense. To avoid starvation, it's important to make sure that every job gets at least one ticket.

Stride

A **stride scheduler** is a deterministic version of a lottery scheduler. Like a lottery scheduler, the stride scheduler gives each task some number of tickets. Then, each task is defined a **stride** which is inversely proportional to the number of tickets. Typically, this is calculated as W/n_i where W is a really big number and n_i is the number of tickets given to task i.

On every time slice, the task with the lowest **pass** is chosen. All tasks start with **pass** equal to 0. When a task is chosen, its pass is incremented by its stride. Hence, a smaller stride will allow a task to run more often.

Linux Completely Fair Scheduler (CFS)

The **Linux Completely Fair Scheduler (CFS)** aims to give each task an equal share of the CPU by giving an illusion that each task executes simultaneously on $1/n$ of the CPU. Since hardware needs to give out CPU in full time slices, the scheduler will track CPU time per task and schedule tasks to match up the average rate of execution. When choosing a task to run, the thread with minimum CPU time will be chosen. To accomplish this efficiently, a heap like scheduling queue is used for efficient (logarithmic with respect to number of tasks) popping and pushing operations.

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. Deadlock is a special form of starvation but has a stronger condition since starvation can end but deadlock cannot.

There are four necessary *but not sufficient* conditions for a deadlock to occur. Eliminating any one of these will eliminate the possibility of a 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.

Detection

The following is a deadlock detection 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.
  request, 2D array where the i-th element is the how many of each of resource thread i is requesting.

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

When the algorithm finishes, the system is said to be deadlocked if there are threads left in unfinished.

Handling

There are four main ways a system can handle a deadlock.

Denial

Pretend deadlock is not a problem (i.e. ostrich algorithm). This is usually effective when deadlocks are uncommon. In the rare times when deadlocks do occur, the system is usually restarted.

Prevention

Write systems that don't result in deadlock. This typically involves eliminating one of the four necessary conditions. Infinite resources could eliminate the bounded resources problem. While not possible for all types of resources, an illusion of infinite resources typically suffices (e.g. virtual memory). No sharing of resources can eliminate mutual exclusion, but totally independent threads are not very realistic and defeats the purpose of using threads. Forcing all threads to request resources in a particular order can also be useful and typically done in many scenarios, but it does require careful coding practices.

Threads could also be forced to not wait and instead keep trying until a successful acquisition of a resource, which is quite inefficient.

Recovery

Let deadlock happen, and recover from it afterwards. A simple and intuitive approach may be to terminate a thread, forcing it to give up its resources. However, this isn't always possible since killing a thread holding a mutex leaves the system in an inconsistent state. The system could also try to take away resources from the thread temporarily, but that may be difficult to fit the semantics of computation. A more general technique is to roll back actions of deadlocked threads. However, this is also prone to deadlock in the same way again if threads are executed in the same order.

Avoidance

Dynamically delay resource requests, so deadlock doesn't happen. The first idea that comes to mind is to check if a resource request would result in a deadlock when a thread requests a resource. However, this may be too late as the system may already be 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. The system must always be kept in a **safe state** where the system can delay resource acquisition requests to prevent deadlock. As a result, the system needs to check if a resource request would result in an *unsafe* state not a deadlocked state.

Banker's algorithm is a deadlock avoidance technique. A thread states the max amount of each resource it needs in advance. The conservative approach would be to allow a particular thread to proceed if the available resources - number requested is at least the max number needed by any other thread. Banker's algorithm takes a less conservative route and pretends each request is granted. Then, it runs the deadlock detection algorithm with an updated condition as seen below.

```
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 thread i will request.

  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
```

The key idea is that banker's algorithm determines if threads are able to be scheduled in a way to avoid deadlock. However, not every execution order has to avoid deadlock.

1.1 Concept Check

1. In what sense is Linux CFS completely fair?

   CFS attempt to give all tasks equal access to the CPU.

2. How can you easily implement lottery scheduling?

Given that each task is allocated their own set number of tickets totaling N tickets across all threads, we can select at random a number between 1 through N. This number would fall into a bin defined by the number of tickets per thread and that thread would then get to run.

3. Is stride scheduling prone to starvation?

No. Stride scheduling tries to achieve proportional shares to the CPU so a thread will eventually get a chance at running. Concretely, if all other threads' pass value are strictly increasing, then even the lowest priority thread will get a chance at running.

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

Smaller stride. Lower stride tasks run more often as their pass value increases at a slower rate.

1.2 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.

```c
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);
}
```

pintos_sps.c

1. Complete the blanks of `thread_unblock` to complete implement SPS. Assume that SPS will treat the ready queue as FIFO. You may ignore any possibilities of preemptions.

If the current thread has a high priority, we place it at the front of the ready queue. Otherwise, we place it at the back.

```c
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);
}
```

pintos_sps_sol.c

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, or gets blocked.

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

The insert operations are cheaper, and it provides a 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.

We can have more than 2 priorities but still a small number of fixed priorities, and have a queue for each priority, and then pop off threads from each queue as necessary.

1.3 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:

Total		
A	B	C
7	8	9

	Current			Max		
T/R	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

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

Yes. To find a safe sequence of executions, we need to first calculate the available resources and the needed resources for each thread. To find the available resources, we sum up the currently held resources from each thread and subtract that from the total resources.

Available		
A	B	C
1	1	1

To find the needed resources for each thread, we subtract the resources they currently have from the maximum they need.

Needed			
	A	B	C
T1	4	1	1
T2	1	4	8
T3	0	1	1
T4	2	0	3

From these, we see that we must run T3 first, as that is the only thread for which all needed resources are currently available. After T3 runs, it returns its held resources to the resource pool, so the available resource pool is now as follows.

Available		
A	B	C
4	1	5

We can now run either T1 or T4, and following the same process, we can arrive at a possible execution sequence of either $T3 \to T1 \to T4 \to T2$ or $T3 \to T4 \to T1 \to T2$.

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

Following the same procedure from the previous question, we see that there are 0 instances of C available at the start of this execution. However, every thread needs at least 1 instance of C to run, so we are unable to run any threads and thus the system is not in a safe state.