Section 7

Wednesday, June 5, 2019 2:39 PM

section7

Section 7: Scheduling and Fairness

October 9-10, 2018

Contents

1 Warmup

Consider infinite quanta

Which of the following are true about Round Robin Scheduling?

Round Robin Scheduling reduces wait time to N * I
time * N / 2. Assuming burst time is small, Round r

T 1. The average wait time is less that that of FCFS for the same workload.

T 2. Is supported by `thread_tick` in Pintos.

T 3. It requires pre-emption to maintain uniform quanta.

T 4. If quanta is constantly updated to become the # of cpu ticks since boot, Round Robin becomes FIFO.

F 5. If all threads in the system have the same priority, Priority Schedulers **must** behave like round robin.

F 6. Cache performance is likely to improve relative to FCFS.

T 7. If no new threads are entering the system all threads will get a chance to run in the cpu every `QUANTA*SECONDS_PER_TICK*NUMTHREADS` seconds. (Assuming `QUANTA` is in ticks).

RF
Pr

T 8. This is the default scheduler in Pintos

Yes, this is the default implementation in Pintos

FCFS h
It will
Cache

T 9. It is the fairest scheduler

Resource allocation in the limit, is uniformly distributed

2 Vocabulary

- **Scheduler** - The process scheduler is a part of the operating system that decides which process runs at a certain point in time. It usually has the ability to pause a running process, move it to the back of the running queue and start a new process;
- **FIFO Scheduling** - First-In-First-Out (aka First-Come-First-Serve) scheduling runs jobs as they arrive. Turnaround time can degrade if short jobs get stuck behind long ones (convoy effect);
- **Round-Robin Scheduling** - Round-Robin scheduling runs each job in fixed-length time slices (quanta). The scheduler preempts a job that exceeds its quantum and moves on, cycling through the jobs. It avoids starvation and is good for short jobs, but context switching overhead can become important depending on quanta length;
- **Priority Scheduling** - Priority scheduling runs the highest priority job, based on some assigned priorities. Starvation of low-priority jobs and priority inversion (a higher priority task waiting for a lower priority one, usually for a lock) are issues. Priorities can be static or dynamic, and if dynamic can change based on heuristics or locking-related donations;
- **SRTF Scheduling** - Shortest Remaining Time First scheduling runs the job with the least remaining amount of computation time and is preemptive. It has the optimally shortest average turnaround time. In practice remaining computation time can't be predicted, so SRTF is often used as a post-facto benchmark for other algorithms;
- **Multi-Level Feedback Queue Scheduling** - MLFQS uses multiple queues with priorities, dropping CPU-bound jobs that consume their entire quanta into lower-priority queues;

burst time. FCFS has an average wait time of process
robin reduces wait titme

Thread tick forces a thread to yield after it consumes it's time slice.

No thread can ticks > ticks since boot,
So the thread will never be pre-empted

R requires pre-emption based on time-slice.
re-emption is not required with priority scheduling (how we pop off the Q)

has ideal cache performance.
consume it's entire cache, and, with most
line / page replacement strategies, always improves with maximal cache size

absolutely

3 Problems

3.1 Scheduling

Consider the following single-threaded processes and their arrival times, CPU bursts, and priority

Process	Arrival Time	CPU Burst	Priority
A	1	5	1
B	3	3	3
C	5	2	2
D	4	4	4

Please note:

- Priority scheduler is preemptive.
- Newly arrived processes are scheduled last for RR. When the RR quanta expires, the currently running thread is added at the end of to the ready list before any newly arriving threads.
- Break ties via priority in Shortest Remaining Time First (SRTF).
- If a process arrives at time x, they are ready to run at the beginning of time x.
- Ignore context switching overhead.
- The quanta for RR is 1 unit of time.
- Total turnaround time is the time a process takes to complete after it arrives.

Fill in the following scheduling table and calculate the total turnaround time for each scheduling algorithm

Time	FIFO	RR	SRTF	Priority
1	A:	A(5):	A(5)	A(5):
2	A:	A(4):	A(4)	A(4)
3	A: B	A(3):B(3)	B(3):A(3)	B(3):A(3)
4	A: BD	B(3):A(2)D(4)	B(2):A(3)D(4)	D(4):B(2)A(3)
5	A: BDC	A(2):D(4)B(2)C(2)	B(1):C(2)A(3)D(4)	D(3):B(2)C(2)A(3)
6	B:DC	D(4):B(2)C(2)A(1)	C(2):A(3)D(4)	D(2):B(2)C(2)A(3)
7	B:DC	B(2):C(2)A(1)D(3)	C(1):A(3):D(4)	D(1):B(2)C(2)A(3)
8	B:DC	C(2):A(1)D(3)B(1)	A(3):D(4)	B(2):C(2)A(3)
9	D:C	A(1):D(3)B(1)C(1)	A(2):D(4)	B(1):C(2)A(3)
10	D:C	D(3):B(1)C(1)	A(1):D(4)	C(2):A(3)
11	D:C	B(1):C(1)D(2)	D(4)	C(1):A(3)
12	D:C	C(1):D(2)	D(3)	A(3)
13	C	D(2)	D(2)	A(2)
14	C	D(1)	D(1)	A(1)
Total Turnaround Time	5+6+10+8=29	9+9+8+11 = 37	10+3+3+11 = 27	14+7+7+4 = 32

30

3.2 Simple Priority Scheduler

We are going to implement a new scheduler in Pintos we will call it SPS. We will just split threads into two priorities "high" and "low". High priority threads should always be scheduled before low priority threads. Turns out we can do this without expensive list operations.

For this question make the following assumptions: TWO PRIORITY LEVELS

- Priority Scheduling is NOT implemented
- High priority threads will have priority 1
- Low priority threads will have priority 0
- The priorities are set correctly and will never be less than 0 or greater than 1
- The priority of the thread can be accessed in the field `int priority` in `struct thread`
- The scheduler treats the ready queue like a FIFO queue READY QUEUE IS FIFO (two ready queues?)
- Dont worry about pre-emption.

Modify `thread_unblock` so SPS works correctly.

You are not allowed to use any non constant time list operations

```
void
thread_unblock (struct thread *t)
{
  enum intr_level old_level;
  ASSERT (is_thread (t));
  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);
}
```

3.2.1 Fairness

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.

Demote after quanta is used (round robin-esque). Blocked or yielded threads are upgraded.

3.2.2 Better than Priority Scheduler?

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?

Let urgent proceses finsih first, but still has built in fairness constraints (round robin like,
Automatic demotion / promotion)
Linked list operations are cheap

3.2.3 Tradeoff

How can we trade off between the coarse granularity of SPS and the super fine granularity of normal priority scheduling? (Assuming we still want this fast insert)

Add more levels

3.3 Totally Fair Scheduler

You design a new scheduler, you call it TFS. The idea is relatively simple, in the begining, we have three values `BIG_QUANTA`, `MIN_LATENCY` and `MIN_QUANTA`. We want to try and schedule all threads every `MIN_LATENCY` ticks, so they can get atleast a little work done, but we also want to make sure they run *atleast* `MIN_QUANTA` ticks. In addition to this we want to account for priorities. We want a threads priority to be inversely proportial to its `vruntime` or the amount of ticks its spent in the CPU in the last `BIG_QUANTA` ticks.

You may make the following assumptions in this problem:

- Priority scheduling in Pintos is functioning properly,
- Priority donation is not implemented.
- Alarm is not implemented.
- `thread_set_priority` is never called by the thread
- You may ignore the limited set of priorities enforced by pintos (priority values may span any `float` value)
- For simplicity assume floating point operations work in the kernel

Over a time period of big_quanta
I want a thread to run at least every min_quanta and for at lea
Min_quanta ticks.

3.3.1 Per thread quanta

How long will a particular thread run? (use the threads priority value)

Max(min_quanta, thread_priority / total_thread_priorities *
min_latency)

In

Th
M

Af

st

the first round, everyone has high priority.

he first running thread starts running and it's priority dips below it's peers, but it must continue to run
in_quanta.

fter running min_quanta

3.3.2 struct thread

Below is the declaration of `struct thread`. What field(s) would we need to add to make TFS possible? You may not need all the blanks.

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    float priority;                       /* Priority, as a float. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif
        Int vruntime;
    ___________________________         /* What goes here? */

        Int quanta;
    ___________________________         /* What goes here? */

    ___________________________         /* What goes here? */

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

3.3.3 thread tick

What is needed for `thread_tick()` for TFS to work properly? You may not need all the blanks.

```
void
thread_tick (void)
{
  struct thread *t = thread_current ();

  /* Update statistics. */
  if (t == idle_thread)
    idle_ticks++;
#ifdef USERPROG
  else if (t->pagedir != NULL)
    user_ticks++;
#endif
  else
    kernel_ticks++;

     T->vruntime++
  ______________________________________________;

  ______________________________________________;
```

T->quanta

```
/* Enforce preemption. */
if (++thread_ticks >= TIME_SLICE) {  /* TIME_SLICE may need to be replaced with something else */
  intr_yield_on_return ();
  T->priority = (1.0/t ->vruntime)  ;
  Float total_priority = 0.0f  ;
  For (e = list_begin (&all_list); e!= list_end (&all_list); e= list_next( e)) {
  ____________________________________;
  Struct thread * t = list_entry (e, sutrct thread allelem);
  ____________________________________;
  Total_priority += t-> priority;
  ____________________________________;
  }  ;
  T-> quanta = max(t->priority/total_priority*MIN_LATENCY, MIN_QUANTA);
 }
}
```

3.3.4 timer interrupt

What is needed for `timer_interrupt` for TFS to function properly.

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  ____________________________________;
     If a big quanta has passed
  ____________________________________;
     Reset everyones runtime and priority
  ____________________________________;
  ____________________________________;
  ____________________________________;
  ____________________________________;
  ____________________________________;
  ____________________________________;
  ____________________________________;
  ____________________________________;
  ____________________________________;
  ____________________________________;
```

```
  thread_tick ();
}
```

3.3.5 thread create

What is needed for `thread_create()` for TFS to work properly? You may not need all the blanks.

```
tid_t
thread_create (const char *name, int priority, thread_func *function, void *aux)
{
  /* Body of thread_create omitted for brevity */
     Old_level = intr_disable()
     Total priority = get_total_priorty()
     For each thread, re-assign it's quanta
     Intr_set_level(old_level);
     Thread_unblock(t);
     If (priority > thread_get_priority())
      thread_yield();
     Return tid
  ------------------------------------------------
  ------------------------------------------------
  ------------------------------------------------
  ------------------------------------------------
  ------------------------------------------------
  ------------------------------------------------
  ------------------------------------------------
  ------------------------------------------------

  /* Add to run queue. */
  thread_unblock (t);
  if (priority > thread_get_priority ())
    thread_yield ();

  return tid;
}
```

3.3.6 Analysis

Explain the high level behavior of this scheduler; what exactly is it trying to do? How is it different/similar from/to the multilevel feedback scheduler from the project?

Very proactive in updating priority.

Lots of expensive computation to re-assign quanta after any priority change