162 Dis 6 Notes

Thursday, February 28, 2019 5:34 PM

Section 6: Deadlock, Scheduling, and Fairness

March 1, 2019

Contents

Detailed Explanations for T/F

1)

Quanta 2

A3 B3 C3

	1	2	3	4	5	6	7	8	9
FCFS	A	A	A	B	B	B	C	C	C
RR	A	A	B	B	C	C	A	B	C

avg wait

A:0 B:3 C:6 → 3 (better)

A: 4 B: 5 C: 6 → 5
(0+4, 2+3, 4+2)

5) Not forced to RR.
Can schedule same thread again (No time quanta requirement)

1 Warmup

Which of the following are true about Round Robin Scheduling?

1. The average wait time is less that that of FCFS for the same workload.
2. Is supported by `thread_tick` in Pintos.
3. It requires pre-emption to maintain uniform quanta.
4. If quanta is constantly updated to become the # of cpu ticks since boot, Round Robin becomes FIFO.
5. If all threads in the system have the same priority, Priority Schedulers **must** behave like round robin.
6. Cache performance is likely to improve relative to FCFS.
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).
8. This is the default scheduler in Pintos
9. It is the fairest scheduler

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;

3 Problems

3.1 Locking Up the Floopies

In section 5, you may remember encountering race conditions inside of the Central Galactic Floopy Corporation's currency exchange server, which runs on top of pthreads. We said that we could make the transactions run correctly by making the balance increment/decrement atomic. The Central Galactic Floopy Corporation hires a consultant named Morty who suggests making the increment/decrement pair appear atomic by adding a lock to each account, and acquiring the locks when we run the transaction.

```
typedef struct account_t {
  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZED;
  int balance;
  long uuid;
};

void transfer(account_t *donor, account_t *recipient, float amount) {
  // lock accounts so we can make the transfer safely
  pthread_mutex_lock(&donor->lock);
  pthread_mutex_lock(&recipient->lock);

  // check balances and make transfer if possible
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
  } else {
    donor->balance -= amount;
    recipient->balance += amount;
  }

  // unlock accounts
  pthread_mutex_unlock(&recipient->lock);
  pthread_mutex_unlock(&donor->lock);
}
```

If we use the locking code given above, will the code run correctly? Has Morty introduced a new bug into our code? Can you give an example of where this code would fail?

Deadlock - like example last week:

1) A → B
2) B → A ← both acquire donor first
1 gets A lock, 2 gets B lock
→ wait on recipient lock (1 wait on B) (2 wait on A)
→ stuck forever

Can you modify the code above to resolve this bug?

```
typedef struct account_t {
  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZED;
  int balance;
  long uuid;
};

void transfer(account_t *donor, account_t *recipient, float amount) {
// lock accounts so we can make the transfer safely
  if (donor->uuid <= recipient->uuid) {
    pthread_mutex_lock(&donor->lock);
    pthread_mutex_lock(&recipient->lock);
  } else {
    pthread_mutex_lock(&recipient->lock);
    pthread_mutex_lock(&donor->lock);
  }


  // check balances and make transfer if possible
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
  } else {
    donor->balance -= amount;
    recipient->balance += amount;
  }

  // unlock accounts
  pthread_mutex_unlock(&recipient->lock);
  pthread_mutex_unlock(&donor->lock);
}
```

always get smaller acc # first

→ global preference order

account lock preferences $1 < 2 < \ldots < n$

3.2 Banker's Algorithm

Covered in detail next week – will do high level explanation

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:

Bankers Algo
- check if we are in a safe state

safe meaning deadlock can be avoided for sure

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

Bankers is conservative
- non safe state ≠ deadlock (mostly like a warning)
- assumes: need max resources to complete, no preemption of resources

Need:

T1	4	1	1
T2	1	4	8
T3	0	1	1
T4	2	0	3

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

0	Available: 7-0-2-3-1=1 8-2-2-3=1 9-2-1-4-1=1
1	Run: T3 → Available: 1+3=4 1+0=1 1+4=5
2	Run: T1 → Available: 4+0=4 1+2=3 5+2=7
3	Run: T4 → Available: 4+1=5 3+3=6 7+1=8 → run T2

Repeat the previous question if the total number of C instances is 8 instead of 9.

0	Available: 1 1 8-2-1-4-1=0
	→ no thread can complete → NOT SAFE

deadlock? maybe, maybe not

what if a thread gives up resource before completing? ⇒ state does not necessarily lead to deadlock.

3.3 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: A→B→D→C

- 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

Arrival	Time	FIFO	RR	Queue @ end of time	SRTF	Time left (A B C D) at beginning	Priority
A	1	A	A	A	A	5	A
	2	A	A	A	A	4	A
B	3	A	A	BA	B	3 3	B
D	4	A	B	ADB	B	3 2 4	D
C	5	A	A	DBCA	B	3 1 4 2	D
	6	B	D	BCAD	C	3 4 2	D
	7	B	B	CADB	C	3 4 1	D
	8	B	C	ADBC	A	3 4	B
	9	D	A	DBC	A	2 4	B
	10	D	D	BCD	A	1 4	C
	11	D	B	CD	D	4	C
	12	D	C	D	D	3	A
	13	C	D	D	D	2	A
	14	C	D		D	1	A
	Total Turnaround Time	30	37		27		32

A B C D
5 + 6 + 9 + 10 (6-1, 9-3, 13-4, 15-5)   9+9+8+11   10+3+3+11   14+7+7+4

finish time - arrival time

3.4 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:

- 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
- 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);
}
```

3.4.1 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.

pri ↓ if uses up quantu, pri ↑ if it yields – prefers smaller bursts

Definition of fair? - all equal time
- important ones go first
...

3.4.2 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?

insert is easier - O(1) inserts
still does a decent job as a priority scheduler
- less "fine grain" details, but still good

3.4.3 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)

happy medium between the two.

3.5 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

3.5.1 Per thread quanta

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

skip - see soln

3.5.2 struct thread

Skip rest, see soln for details.

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
    ___________________________         /* What goes here? */

    ___________________________         /* What goes here? */

    ___________________________         /* What goes here? */

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

3.5.3 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++;

  _____________________________________________;

  _____________________________________________;
```

```
  /* Enforce preemption. */
  if (++thread_ticks >= TIME_SLICE) {  /* TIME_SLICE may need to be replaced with something else */
    intr_yield_on_return ();

    _________________________________;

    _________________________________;

    _________________________________;

    _________________________________;

    _________________________________;

    _________________________________;

    _________________________________;
  }
}
```

3.5.4 timer interrupt

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

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  _____________________________________;

  _____________________________________;

  _____________________________________;

  _____________________________________;

  _____________________________________;

  _____________________________________;

  _____________________________________;

  _____________________________________;

  _____________________________________;

  _____________________________________;

  _____________________________________;

  _____________________________________;
```

```
  thread_tick ();
}
```

3.5.5 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)
{
  /* Block of thread_create omitted for brevity */

 ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________


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

  return tid;
}
```

3.5.6 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?