SchedLab Report

策略分析

1）$FCFS$

- 算法实现：

  使用$First\ Come\ First\ Serve$策略进行调度。

  用两个队列维护当前需要使用CPU的任务的等待队列和当前需要进行IO的任务的等待队列。每次对队头的任务进行CPU处理和IO操作。

  当**新任务到达**时，将其加入了CPU的等待队列。当**一个任务需要进行**IO时，将这个任务从CPU队列的队头删除（该任务一定是当前CPU处理的任务），然后加入到IO队列等待进行IO操作。当**一个任务完成**IO**操作**时，将其从IO队列的对头删除（该任务一定时当前进行IO操作的任务），并加入到CPU等待队列。当**一个任务完成**时，由于约定了任务一定以CPU操作作为结尾，所以当前完成的任务一定在CPU队列的队头，将其删除即可。

  每次调度器被唤醒的时候，不对当前进行CPU处理和IO处理的任务进行抢占。除非当前任务已经结束，那么在对应队列非空的情况下选择队头进行相应的操作。

- 优点：算法实现较为简单。
- 缺点：任务完成率较低，可能会被运行时间长的任务长时间占用CPU和IO处理。
- 得分：40。
- 代码实现：

```
Action policy(const std::vector<Event>& events, int current_cpu,
int current_io) {
    for (auto e : events) {
        if (e.type == Event::Type::kTimer) {
            continue;
        } // 遇到一个时间片，不进行任何处理
        else if (e.type == Event::Type::kTaskArrival) {
            que_cpu.push(e.task.taskId);
        } // 遇到一个新的任务，将其加入到CPU队列中
        else if (e.type == Event::Type::kTaskFinish) {
            if (e.task.taskId == que_cpu.front())
```

```
                que_cpu.pop();
                // 将完成的任务弹出队列
            else
                fprintf (stderr, "wrong task finish!\n");
                // 如果完成的不是期望的任务，进行报错。
        }
        else if (e.type == Event::Type::kIoRequest) {
            if (e.task.taskId == que_cpu.front()) {
                que_cpu.pop();
                que_io.push(e.task.taskId);
                // 当一个任务需要进行IO时，将其从CPU队列弹出，加入到IO队
列
            }
            else
                fprintf (stderr, "wrong task requests IO!\n");
        }
        else if (e.type == Event::Type::kIoEnd) {
            if (e.task.taskId == que_io.front()) {
                que_io.pop();
                que_cpu.push(e.task.taskId);
                // 当一个任务完成IO操作时，将其放回CPU队列
            }
            else
                fprintf (stderr, "wrong task finish IO!\n");
        }

    }
    int x = 0, y = 0; // 处理化为空转
    if (!que_cpu.empty()) x = que_cpu.front();
    if (!que_io.empty()) y = que_io.front();
    // 分别选取队首的任务进行处理
    return Action{x, y};
}
```

2）RR

- 算法实现：

  使用类*Round Robin*策略进行调度。

  与*FCFS*策略类似，使用两个队列维护当前需要使用*CPU*的任务的等待队列和当前需要进行*IO*的任务的等待队列。每次对队头的任务进行*CPU*处理和*IO*操作。

对于新任务到达、任务完成、任务需要进行IO操作，任务完成IO操作这四种情况，处理方法与$FCFS$时进行相同的处理。当一个时间片结束时，将当前的任务（如果仍然未完成的话）放到CPU队列的末尾。对于IO操作的队列，由于IO操作不可打断，所以不对其进行特别的操作。

- 优点：算法实现较为简单，可以让每个任务都得到较为均衡的运行时间，运行时间较短的任务不会被阻塞。
- 缺点：对于所有任务需要的时间都较长的情况，可能导致多个任务都无法完成，平均周转时间较长。
- 得分：63。
- 核心代码：

```
    ......
        if (e.type == Event::Type::kTimer) {
            if (events.size() != 1) continue;
            // 为了方便代码的编写，当只进行时间片操作时才进行处理。
            if (!que_cpu.empty()) {
                int x = que_cpu.front();
                // 从队头取出任务
                que_cpu.pop();
                que_cpu.push(x);
                // 将该任务放入队尾
            }
        }
    ......
```

3）RR改进版

- 算法实现：

  在$Round\ Robin$的基础上进行改进，在每个时间片到来的时候，对于每个任务，在94%的概率下，当$deadline$和$arrivetime$相差较小（高优先级与低优先级的界限不同）的情况下，不会将其放置到队尾，而是保持该任务在队头，使其能够在下一个时间片内继续执行。

  程序中的各个参数均为测试得到、表现较好的参数。当截至时间与到达时间相差较小时，将认为这个任务的时间较为紧迫，需要尽量早的处理这个任务以按时完成，同时这个任务可能运行时间较短，尽早处理可以完成尽量多的任务。
- 优点：优先完成高优先级，且更为时间较为紧迫的任务。
- 缺点：对于所有任务需要的时间都较长的情况，可能导致多个任务都无法完成，平均周转时间较长。
- 得分：67

- 核心代码：

```
const int propH = 94;
const int lengthH = 30000;
const int propL = 94;
const int lengthL = 20000;

    ......
    if (e.type == Event::Type::kTimer) {
            if (events.size() != 1) continue;
            if (!que_cpu.empty()) {
                Event::Task x;
                x = que_cpu.front();

                if (x.priority == Event::Task::Priority::kHigh
&&
                    rand() % 100 < propH &&
                    x.deadline - x.arrivalTime <= lengthH)
continue;

                if (x.priority == Event::Task::Priority::kLow &&
                    rand() % 100 < propL &&
                    x.deadline - x.arrivalTime <= lengthL)
continue;
                // 如果当前任务是低优先级任务，

                que_cpu.pop();
                que_cpu.push(x);
            }
        }
    ......
```

4）$MLFQ$

- 算法实现：

  使用$Multi-Level\ Feedback\ Queue$策略进行调度。

  当**每个时间片到达**的时候，将当前的任务防止到下一个优先级队列中，并且标记当前可以进行一个新任务。当**一个新任务到达**时，将这个任务加入最高的优先级队列中。当**一个任务完成**时，将其从队列中删除，并且标记当前可以进行一个新任务。当**一个任务需要进行**IO时，将这个任务从CPU队列中弹出，并加入到IO队列，并且标记当前可以进行一个新任务。由于IO操作不可被抢占，所以只需要

设置一个*IO*队列即可。当一个任务完成*IO*时，将其从*IO*队列弹出，并且加入到*CPU*的最高优先级队列。

当可以进行新任务时，选择当前优先级最高的队列的队头进行处理。否则维持*CPU*和*IO*处理原来的任务。

- 优点：能够较好的处理长运行时间的任务，能够使得较多的短时间的任务较快完成，以提高任务的完成率。
- 缺点：没有考虑任务的优先级之分，导致长时间的高优先级任务可能会被放到队列的后部而没能完成。
- 得分：71
- 核心代码：

```
Action policy(const std::vector<Event>& events, int current_cpu,
int current_io) {

    srand(19260817);

    bool change = false;

    for (auto e : events) {
        if (e.type == Event::Type::kTimer) {
            sumtp++;
            change = true; // 标记可以当前修改处理的任务
            if (events.size() != 1) continue;
            for (int pos = 0; pos < nque; ++pos) {
                if (!que_cpu[pos].empty()) {
                    Event::Task x;
                    x = que_cpu[pos].front();
                    if (x.taskId != current_cpu) continue;
                    // 如果当前队列的队头不是处理中的任务则跳过
                    que_cpu[pos].pop();
                    if (pos + 1 < nque) pos++;
                    // 放到下一个优先级队列
                    que_cpu[pos].push(x);
                    break;
                }
            }
        }
        else if (e.type == Event::Type::kTaskArrival) {
            que_cpu[0].push(e.task);
            // 将新任务添加到最高优先级的队列
        }
```

```
30          else if (e.type == Event::Type::kTaskFinish) {
31              bool flag = false;
32              for (int pos = 0; pos < nque; ++pos) {
33                  if (que_cpu[pos].empty()) continue;
34                  if (e.task.taskId ==
    que_cpu[pos].front().taskId) {
35                      if (current_cpu != 0)
36                          fprintf (stderr, "wrong task has
    finished!");
37                      que_cpu[pos].pop();
38                      flag = true;
39                      change = true;
40                      break;
41                  }
42              } // 找到相应的任务，将其弹出队列，并处理标记
43              if (!flag)
44                  fprintf (stderr, "wrong task has finished!\n");
45          }
46          else if (e.type == Event::Type::kIoRequest) {
47              bool flag = false;
48              for (int pos = 0; pos < nque; ++pos) {
49                  if (que_cpu[pos].empty()) continue;
50                  if (e.task.taskId ==
    que_cpu[pos].front().taskId) {
51                      if (current_cpu != 0)
52                          fprintf (stderr, "wrong task requests
    IO!\n");
53                      que_cpu[pos].pop(); // 从CPU队列弹出任务
54                      que_io.push(e.task); // 加入到IO队列中
55                      flag = true;
56                      change = true;
57                      break;
58                  }
59              }
60              if (!flag)
61                  fprintf (stderr, "wrong task requests IO!\n");
62          }
63          else if (e.type == Event::Type::kIoEnd) {
64              if (e.task.taskId == que_io.front().taskId) {
65                  if (current_io != 0)
66                          fprintf (stderr, "wrong task has
    finished IO!\n");
67                  que_io.pop(); // 从IO队列中弹出
```

```
68                  que_cpu[0].push(e.task);
69                  // 加入到最高优先级队列
70              }
71              else
72                  fprintf (stderr, "wrong task has finished
    IO!\n");
73          }
74
75      }
76
77      int x = 0, y = 0;
78      if (change) { // 如果当前可以更改处理的任务才进行更改
79          for (int pos = 0; pos < nque; ++pos) {
80              if (!que_cpu[pos].empty()) {
81                  x = que_cpu[pos].front().taskId;
82                  break;
83              }
84          }
85      }
86      else x = current_cpu;
87      if (!que_io.empty()) y = que_io.front().taskId;
88      return Action{x, y};
89  }
```

5）$MLFQ$改进算法

- 算法实现：

  对$Multi-Level\ Feedback\ Queue$策略进行一定程度上的改进。

  在**时间片到达**的时候，如果当前任务是高优先级任务，那么将其降低一个等级。如果当前任务是低优先级任务，将其降低三个等级。同时**取消了可修改标记**，每次调度器被唤醒的时候，都可以在当前最高优先级队列中选取队头进行处理，不需要等到时间片或者当前任务完成才处理新的任务。最后，对于**已经超过截至时间的任务**，主档放弃这些任务，将其加入到另一个队列中，这个队列任务只有在**没有还未达到截至时间的任务可以处理的时候**会被处理。

- 优点：能够简单地区分高优先级任务和低优先级任务，可以提高高优先级任务的完成率，同时取消了可修改标志，可以提高短任务的完成率。主动放弃超时的任务，可以提高其他任务的完成率。

- 缺点：对于截至时间较为紧迫的任务没有良好的处理结果，由于不断地调度科恩那个会出现所有任务都无法完成的情况。

- 得分：80

- 核心代码：

```
Action policy(const std::vector<Event>& events, int current_cpu, int current_io) {

    srand(19260817);
    int ti = 0;

    for (auto e : events) {
        ti = e.time;
        if (e.type == Event::Type::kTimer) {
            sumtp++;
            if (events.size() != 1) continue;
            for (int pos = 0; pos < nque; ++pos) {
                if (!que_cpu[pos].empty()) {
                    Event::Task x;
                    x = que_cpu[pos].front();
                    if (x.taskId != current_cpu) continue;
                    que_cpu[pos].pop();
                    if (pos + 1 < nque - 1) pos++;
                    if (x.priority == Event::Task::Priority::kLow) {
                        if (pos + 1 < nque - 1) pos++;
                        if (pos + 1 < nque - 1) pos++;
                    } // 对于低优先级任务进行特殊处理
                    que_cpu[pos].push(x);
                    break;
                }
            }
        }
        else if (e.type == Event::Type::kTaskArrival) {
            que_cpu[0].push(e.task);
        } // 将新任务加到最高优先级队列
        else if (e.type == Event::Type::kTaskFinish) {
            bool flag = false;
            for (int pos = 0; pos < nque; ++pos) {
                if (que_cpu[pos].empty()) continue;
                if (e.task.taskId == que_cpu[pos].front().taskId) {
                    if (current_cpu != 0)
                        fprintf (stderr, "wrong task has finished!");
                    que_cpu[pos].pop();
```

```
                    flag = true;
                    break;
                }
            } // 找到并弹出完成的任务
            if (!flag)
                fprintf (stderr, "wrong task has finished!\n");
        }
        else if (e.type == Event::Type::kIoRequest) {
            bool flag = false;
            for (int pos = 0; pos < nque; ++pos) {
                if (que_cpu[pos].empty()) continue;
                if (e.task.taskId ==
que_cpu[pos].front().taskId) {
                    if (current_cpu != 0)
                        fprintf (stderr, "wrong task requests
IO1!\n");
                    que_cpu[pos].pop();
                    que_io.push(e.task);
                    flag = true;
                    break;
                }
            } // 找到任务并将其加入到IO队列中
            if (!flag)
                fprintf (stderr, "wrong task requests IO2!\n");
        }
        else if (e.type == Event::Type::kIoEnd) {
            if (!que_io.empty() && e.task.taskId ==
que_io.front().taskId) {
                if (current_io != 0)
                        fprintf (stderr, "wrong task has
finished IO1!\n");
                que_io.pop();
                que_cpu[0].push(e.task);
            } // 将完成IO的任务放到CPU最高优先级队列
            else if (e.task.taskId ==
que_io_ddl.front().taskId) {
                if (current_io != 0)
                        fprintf (stderr, "wrong task has
finished IO1!\n");
                que_io_ddl.pop();
                que_cpu[nque - 1].push(e.task);
            } // 处理已超时的任务
            else
```

```
                fprintf (stderr, "wrong task has finished IO2!\n");
        }

    }

    int x = 0, y = current_io;
    for (int pos = 0; pos < nque; ++pos) {
        while (!que_cpu[pos].empty() && pos != nque - 1) {
            if (que_cpu[pos].front().deadline <= ti) {
                Event::Task x;
                x = que_cpu[pos].front();
                que_cpu[pos].pop();
                que_cpu[nque - 1].push(x);

            } // 将已超时的任务放置到最低优先级队列
            else break;
        }
        if (!que_cpu[pos].empty()) {
            x = que_cpu[pos].front().taskId;
            break;
        } // 选择当前优先级最高的队列的队头任务进行处理
    }
    while (!que_io.empty() && current_io == 0) {
        if (que_io.front().deadline <= ti && current_io == 0) {
            Event::Task x;
            x = que_io.front();
            que_io.pop();
            que_io_ddl.push(x);
        } // 处理超时的IO操作
        else break;
    }
    if (!current_io){
        if (!que_io.empty()) y = que_io.front().taskId;
        else if (!que_io_ddl.empty()) y = que_io_ddl.front().taskId;
    } // 选择进行IO操作的任务
    return Action{x, y};
}
```

6）截止时间优先

- 算法实现：

  使用**优先队列**对对任务进行维护。在队列中，使用截止时间进行排序，截止时间早的任务先执行。当一个任务的截至时间已经临近，可以**认为**这个任务已经完成了很大的一部分，接下来使用较短时间即可完成这个任务。同时，截至时间临近的任务较为紧迫，需要尽早完成这个任务。

  同时如果该任务已经超过截至时间，那么主动放弃这个任务。

- 优点：对截至时间临近的任务给予了很高的权重，时间充裕的任务被放到较低优先级，时间紧迫的任务抓紧完成。
- 缺点：没有考虑高优先级和低优先级的影响。
- 得分：86
- 核心代码：

```
int sumtp = 0;

struct cmp{
    bool operator () (Event::Task &i, Event::Task &j) {
        if (i.deadline != j.deadline) return i.deadline > j.deadline;
        else return (i.priority == Event::Task::Priority::kHigh);
    }
};

priority_queue < Event::Task, vector <Event::Task>, cmp > que_cpu, que_io, que_cpu_ddl, que_io_ddl;
Event::Task ccpu, cio;
bool ucpu = false;

Action policy(const std::vector<Event>& events, int current_cpu, int current_io) {

    int ti = 0;
    bool isFinish = false;
    for (auto e : events) {
        ti = e.time;
        if (e.type == Event::Type::kTimer) {
            continue;
        }
        else if (e.type == Event::Type::kTaskArrival) {
```

```
            que_cpu.push(e.task);
        } // 加入新任务
        else if (e.type == Event::Type::kTaskFinish) {
            isFinish = true;
        } // 标记任务已经完成
        else if (e.type == Event::Type::kIoRequest) {
            isFinish = true;
            que_io.push(ccpu);
        } // 加入到IO队列，标记当前课更换任务
        else if (e.type == Event::Type::kIoEnd) {
            que_cpu.push(cio);
            continue;
        } // 加入到CPU队列
    }

    if (!isFinish && ucpu) {
        que_cpu.push(ccpu);
    }
    else {
        ucpu = false;
    }

    int x = 0, y = current_io;

    while (!que_cpu.empty()) {
        if (que_cpu.top().deadline <= ti) {
            Event::Task e = que_cpu.top();
            que_cpu.pop();
            que_cpu_ddl.push(e);
        }
        else break;
    } // 将超时的任务取出
    if (!que_cpu.empty()) {
        ccpu = que_cpu.top();
        que_cpu.pop();
        x = ccpu.taskId;
        ucpu = true;
    } // 处理未超时的任务
    else if (!que_cpu_ddl.empty()) {
        ccpu = que_cpu_ddl.top();
        que_cpu_ddl.pop();
        x = ccpu.taskId;
        ucpu = true;
```

```
67      } // 当没有未超时任务时，才处理超时的任务
68
69      while (!que_io.empty()) {
70          if (que_io.top().deadline <= ti) {
71              Event::Task e = que_io.top();
72              que_io.pop();
73              que_io_ddl.push(e);
74          }
75          else break;
76      }
77      if (current_io == 0) {
78          if (!que_io.empty()) {
79              cio = que_io.top();
80              que_io.pop();
81              y = cio.taskId;
82          }
83          else if (!que_io_ddl.empty()) {
84              cio = que_io_ddl.top();
85              que_io_ddl.pop();
86              y = cio.taskId;
87          }
88      } // IO队列的操作同理
89      return Action{x, y};
90  }
```

7）任务期限长度优先

- 算法实现：

  使用优先队列维护待处理的任务。在优先队列中，对任务期限（$Deadline - ArrivalTime$）进行排序，任务期限短的放在前面。可以认为任务期限较短的任务时间较为紧迫，同时需要进行的处理也更少，更容易完成。

  对于超过截至时间的任务，主动对其进行放弃。同时，只有当一定数量的任务被完成后，且IO空载的时候，才会处理对应的超时的任务。由于CPU处理的任务可以随时被打断，所以当CPU空载时可以马上处理超时的任务。

- 优点：对短任务的完成率较好。同时，推迟了对于超时的任务的处理，使超时的任务在空闲时间才进行处理，以提高其他任务的完成率。

- 缺点：没有考虑高优先级和低优先级对任务的影响。同时，对于任务期限较短且处理时间较长的任务，很可能会浪费CPU和IO资源，即出现对其进行了长时间的处理，但是最后仍然没有完成的情况。

- 得分：89

- 核心代码：

```
int maxn = 0;

namespace XuanXue {

    int sumtp = 0;
    int sum = 0;
    bool flag = false;
    Event::Task ccpu, cio;
    bool ucpu = false;
    int id[200];

    struct cmp{
        bool operator () (Event::Task &i, Event::Task &j) {
            return i.deadline - i.arrivalTime > j.deadline - j.arrivalTime;
        } // 对任务期限进行排序
    };

    priority_queue < Event::Task, vector <Event::Task>, cmp > que_cpu, que_io;
    priority_queue < Event::Task, vector <Event::Task>, cmp > que_cpu_ddl, que_io_ddl;

    Action policy(const std::vector<Event>& events, int current_cpu, int current_io) {

        int ti = 0;
        bool isFinish = false;
        for (auto e : events) {
            ti = e.time;
            if (e.type == Event::Type::kTimer) {
                continue;
            }
            else if (e.type == Event::Type::kTaskArrival) {
                que_cpu.push(e.task);
            } // 加入新任务
            else if (e.type == Event::Type::kTaskFinish) {
                isFinish = true;
                sum++;
            } // 标记任务已经完成
            else if (e.type == Event::Type::kIoRequest) {
```

```
38                  isFinish = true;
39                  que_io.push(ccpu);
40              } // 加入到IO队列，标记当前课更换任务
41              else if (e.type == Event::Type::kIoEnd) {
42                  que_cpu.push(cio);
43                  continue;
44              } // 加入到CPU队列
45          }
46
47          if (!isFinish && ucpu) {
48              que_cpu.push(ccpu);
49          }
50          else {
51              ucpu = false;
52          } // 将当前CPU处理的任务放回优先队列
53
54          int x = 0, y = current_io;
55
56          while (!que_cpu.empty()) {
57              if (que_cpu.top().deadline <= ti) {
58                  Event::Task e = que_cpu.top();
59                  que_cpu.pop();
60                  que_cpu_ddl.push(e);
61              }
62              else break;
63          } // 将超时的任务取出
64          if (!que_cpu.empty()) {
65              ccpu = que_cpu.top();
66              que_cpu.pop();
67              x = ccpu.taskId;
68              ucpu = true;
69          } // 处理未超时的任务
70          else if (!que_cpu_ddl.empty()) {
71              ccpu = que_cpu_ddl.top();
72              que_cpu_ddl.pop();
73              x = ccpu.taskId;
74              ucpu = true;
75          } // 当没有未超时任务时，才处理超时的任务
76
77          while (!que_io.empty()) {
78              if (que_io.top().deadline <= ti) {
79                  Event::Task e = que_io.top();
80                  que_io.pop();
```

```
                que_io_ddl.push(e);
            }
            else break;
        }
        if (current_io == 0) {
            if (!que_io.empty()) {
                cio = que_io.top();
                que_io.pop();
                y = cio.taskId;
            }
            else if (!que_io_ddl.empty() && sum >= maxn) {
                cio = que_io_ddl.top();
                que_io_ddl.pop();
                y = cio.taskId;
            } // 当没有未超时任务且完成一定任务量后，才处理超时的任务
        }
        return Action{x, y};
    }
}

Action policy(const std::vector<Event>& events, int
current_cpu, int current_io) {
    maxn = 15;
    return XuanXue::policy(events, current_cpu, current_io);
}
```

8）数据分治

通过OJ的评测记录可以查看每个数据的第一行，可以得到第一个任务的基本信息。通过判断第一个任务的*deadline*可以对数据进行分治。这种算法可以得到90分的成绩。但是由于该算法**不符合要求**，故这里不做详细说明。

总结

该实验一共使用了六种策略进行测试，这些策略都没有针对特定的数据类型、任务分布情况进行优化。可以发现，**任务期限长度优先策略**在测试中能够得到最高的分数。