Ready, blocked, or done

Johan Montelius and Amir H. Payberah

1 Introduction

Your task is to simulate a scheduling algorithm as we refine it from something that does hardly anything to something that is almost useful.

2 Jobs and queues

Our scheduler will keep track of a set of jobs that will be in either of three lists: `ready`, `blocked`, and `done`. All jobs will be created from the start and placed on the list of blocked jobs. In each iteration, the simulator will:

- unblock jobs that are ready to run by moving them from the `blocked` queue to the `ready` queue,
- schedule one job from the `ready` queue for execution, and
- depending on the result, move it to either of the three lists.

The jobs that the simulator will handle will not do anything. The simulator only knows how much time a job needs to complete its tasks and how often it will do an I/O operation as we extend the simulator. The simulator will collect statistics and, in the end, be able to answer questions such as turnaround time and response time for each job.

We define a structure to represent jobs, and we will keep it simple to start with. The only properties we are interested in right now are arrival time, unblock time, total execution time, and the ratio of I/O operations.

```c
typedef struct job {
  int id;
  int arrival;
  int unblock;
  int exectime;
  float ioratio;
  struct job *next;
} job;
```

The unblock time will initially be set to a value that specifies when the job arrives in the system. If the job is blocked due to an I/O operation, the value will indicate when the operation is completed, and the job could again be scheduled. We will create a unique identifier when we initiate the simulation. To quickly set up a benchmark, we also define a structure describing the different jobs we should use.

```
typedef struct spec {
  int arrival;
  int exectime;
  float ioratio;
} spec;
```

The initialization could then look as follows; we create a global array with 10 jobs and one dummy specification in the end. The jobs could be given in any order, although they are here listed in the order they arrive.

```
spec specs[] = {
  {  0,  10, 0.0},
  {  0,  30, 0.7},
  {  0,  20, 0.0},
  { 40,  80, 0.4},
  { 60,  30, 0.3},
  {120,  90, 0.3},
  {120,  40, 0.5},
  {140,  20, 0.2},
  {160,  10, 0.3},
  {180,  20, 0.3},
  {0,     0,   0}  // dummy job
};
```

We have the three queues as global pointers, which are all null pointers to start with.

```
job *readyq = NULL;
job *blockedq = NULL;
job *doneq = NULL;
```

The simulator should first go through the specifications and create jobs that are added to the queue of blocked jobs.

```
void init() {
  int i = 0;
  while (specs[i].exectime != 0) {
    job *new = (job *)malloc(sizeof(job));
    new->id = i + 1;
    new->arrival = specs[i].arrival;
    new->unblock = specs[i].arrival;
    new->exectime = specs[i].exectime;
    new->ioratio = specs[i].ioratio;
    block(new);
    i++;
  }
}
```

When we add the jobs to the `blocked` queue, we order them so that the jobs with the lowest arrival time are first.

```
void block(job *this) {
  job *nxt = blockedq;
  job *prev = NULL;

  while (nxt != NULL) {
    if (this->unblock < nxt->unblock)
      break;
    else {
      prev = nxt;
      nxt = nxt->next;
    }
  }
  this->next = nxt;
  if (prev != NULL)
    prev->next = this;
  else
    blockedq = this;

  return;
}
```

The simulator will keep track of time and move jobs from the `blocked` queue to the `ready` queue. Since the jobs are ordered, we do not have to search the whole queue. A printout will trace what is happening.

```
void unblock(int time) {
  while (blockedq != NULL && blockedq->unblock <= time) {
    job *nxt = blockedq;
    blockedq = nxt->next;
    printf("(%4d)unblock job %2d\n", time, nxt->id);
    ready(nxt);
  }
}
```

In the first run, we simply add jobs at the end of the `ready` queue. This might not be optimal, but why complicate things.

```
void ready(job *this) {
  job *nxt = readyq;
  job *prev = NULL;

  while (nxt != NULL ) {
    prev = nxt;
    nxt = nxt->next;
  }
  this->next = nxt;
  if (prev == NULL)
    readyq = this;
  else
    prev->next = this;

  return;
}
```

When jobs have terminated, they are added to the `done` queue. The order in this queue is not important, so we might as well add them to the beginning.

```
void done(job *this) {
  this->next = doneq;
  doneq = this;
  return;
}
```

3 The scheduler

The scheduler's job is to select the first job from the `ready` queue and let it "execute". Since this is only a simulation and the jobs do not do anything, we simply set the remaining execution time to zero and move it to the list of terminated jobs. Note - we now allow jobs to execute until they end; there is no preemption.

```
int schedule(int time) {
  if (readyq != NULL) {
    job *nxt = readyq;
    readyq = readyq->next;

    int exect = nxt->exectime;
    nxt->exectime = 0;
    printf("(%4d)run job %2d for %3d ms\n", time, nxt->id, exect);
    done(nxt);
    return exect;
  } else
    return 1;
}
```

The `schedule()` procedure returns how long time has passed, and for this simple scheduler, it is the total execution time of the job. If there is no job in the `ready` queue, it returns 1 to drive the simulation forward (why would the `ready` queue be empty?). The procedure does not need to know the time right now, but we will need it later, and for now, we can use it to do the printout.

So now we have all the pieces to the puzzle and can create our first scheduler and take it for a spin. Remember to include the proper header files in the beginning and that you need to order the procedures (or declare them) so that you do not use a procedure before it is declared.

```
int main() {
  init();
  int time = 0;

  while (blockedq != NULL || readyq != NULL) {
    unblock(time);
    int tick = schedule(time);
    time += tick;
  }

  printf("\ntotal execution time is %d \n", time);

  return 0;
}
```

I hope it worked. Now let's add some more features to the scheduler.

4 Shortest job first

We might first try to adopt the *shortest job first* strategy. Everything will look the same, but when we add jobs to the `ready` queue, we order them so that the shortest jobs occur first. If you add a test to the `ready()` while-loop, you should be able to do it in no time. The result is not immediately visible since the total execution time is the same. What could have improved in the average turnaround time? We add a field to the job structure to check if this is the case.

```
typedef struct job {
  :
  int turnaround;
  :
} job;
```

In the `schedule()` procedure we now calculate the turnaround time.

```
  :
  nxt->turnaround = time + exect - nxt->arrival;
  :
```

We can run through all the jobs in the `done` queue in the main procedure and collect the average turnaround time.

```
  :
  int turnaround = 0;
  int jobs = 0;

  for (job *nxt = doneq; nxt != NULL; nxt = nxt->next) {
    jobs += 1;
    turnaround += nxt->turnaround;
  }

  printf("\naverage turnaround: %d \n", turnaround/jobs);
  :
```

Try with and without ordering the `ready` queue, any difference?

5 Preemptive scheduling

What is the *response time* for each job? The response time is the time from arrival until scheduled so let's keep track of this. We add another field to the job structure.

```
typedef struct job {
  :
  int respons;
  :
} job;
```

Then we ensure that this field is zero when we create new jobs in the `init()` procedure. We set it to zero in the initialization, and when the job is scheduled, we calculate the respons time. In the end, we can calculate the average response time for all jobs. Is there something we can do to improve the response time? How about moving to a preemptive scheduler? Let's provide an argument to the simulator that sets the *time slot* that we will give each job. We then pass this parameter to the `schedule()` procedure.

```
int main(int argc, char *argv[]) {
  int slot = 10;

  if (argc == 2) {
    slot = atoi(argv[1]);
  }
  :
  while (blockedq != NULL || readyq != NULL) {
    :
    int tick = schedule(time, slot);
    :
  }
  :
```

The execution time of a job will be updated to hold the *time to completion*. Every time we schedule a job, we first check if the time to completion is less or equal to the time slot given. If this is so, the code is very similar to before. Otherwise, if the time to completion is longer than the time slot, we should decrease the time and return the job to the `ready` queue. We will use zero to indicate that the response time has not yet been set (this is something we will use later)

In the `schedule()` procedure, we can now check if it is the first time that the job is scheduled, and if so, fill in the response time as the current time minus the arrival time. Note that we now will have to check if the response time has already been set. If the response time is still zero, we know it is the first time the job is scheduled, and we update the value. This is what it could look like:

```
:
if(nxt->respons == 0)
nxt->respons = time - nxt->arrival;

int left = nxt->exectime;
int exect = (left < slot) ? left : slot;

nxt->exectime -= exect;
printf("(%4d)run job %2d for %3d ms ", time, nxt->id, exect);

if (nxt->exectime == 0) {
  nxt->turnaround = time + exect - nxt->arrival;
  printf(" -  done\n");
  done(nxt);
} else {
  ready(nxt);
  printf("- %3d left \n", nxt->exectime);
}

return exect;
```

If you try a time slot of 100, everything will work as before since no job has an execution time greater than 90. If you decrease the time slot, you will see more scheduling events, and you might see the response time improve.

6 I/O operations

So now, for the last problem, what happens if a job performs an I/O operation? Let's say that an I/O operation takes 30 ms to complete, and some jobs will do an operation once in a while. We will extend our simulation to handle I/O operations and then try a smarter scheduler. We assume that all I/O operations take 30 ms and define this as a macro. We also write a small routine that will flip a coin and determine if, given the I/O ratio of the job, an I/O operation is performed. If no I/O operation is performed, it returns zero; otherwise, it returns how long time passes before the operation happens (from 1 to exect -1). We will not use this information now but need it later.

```c
#define IO_TIME 30

int io_op(float ratio, int exect) {
  int io = ((float)rand()) / RAND_MAX < ratio;
  if( io )
    io = (int)trunc(((float)rand()) / RAND_MAX * (exect - 1)) + 1;
  return io;
}
```

To compile this, you need to include `stdlib.h` and compile with the math library using the flag `-lm`. Once this is in place, we can modify the `schedule()` procedure. Note that we are not doing anything else while the job is doing the I/O operation. The procedure will simply return the execution time plus the time it took to do the I/O operation.

```c
  :
  int io = 0;

  if (exect > 1)
      io = io_op(nxt->ioratio, exect);

  nxt->exectime -= exect;
  printf("(%4d) run job %2d for %3d ms ", time, nxt->id, exect);

  if (nxt->exectime == 0) {
    nxt->turnaround = time + exect - nxt->arrival;
    printf("-  done\n");
    done(nxt);
  } else {
    if (io) {
      ready(nxt);
      printf("- %3d left - I/O \n", nxt->exectime);
      exect += IO_TIME;
    } else {
      ready(nxt);
      printf(" - %3d left \n", nxt->exectime);
    }
  }

  return exect;
```

Give it a try and see what happens. The total execution time will probably increase, which is quite understandable. We are sitting around waiting for I/O operations. Is there something we can do?

7 Block jobs until I/O completed

A CPU is basically doing nothing while an I/O operation is done. To just sit around and wait is a complete waste of time. If we could schedule another job in the meantime, we would improve the situation. How about moving jobs back to the `blocked` queue until they are ready to execute again? It turns out that we have all pieces to the puzzle to do this so we only have to change the scheduler slightly. First we set the execution time to whatever the `io_op()` function returns (if it is different from zero).

```
if (exect > 1) {
  io = io_op(nxt->ioratio, exect);
  if (io)
    exect = io;
}
```

Then we change what we do if an I/O operation is issued. We set the `unblock` value of the job and return it to the queue of blocked jobs instead of the `ready` queue. We also patch the printout to see what is happening.

```
if (io) {
  nxt->unblock = time + exect + IO_TIME;
  block(nxt);
  printf("- %3d left - blocked\n", nxt->exectime);
} else {
  ready(nxt);
  printf("- %3d left\n", nxt->exectime);
}
```

8 What's more?

The scheduler we have now is still very simple. It keeps all jobs that are ready to execute in one queue. The queue is ordered so that jobs with a short completion time will be handled first. However, one can question if this is always relevant or if this information is known. If we do not know the time to completion, which job should we select for execution? Should we have jobs with different priorities, or should we have several queues? If you extend this scheduler, you will soon end up with something that looks like a *multi-level feedback queue scheduler* where jobs that do I/O operations are given priority.