Systèmes d'exploitation

TP 3 - Ordonnancement

2023-03-09

Cet exercice provient d'un devoir proposé par ***Johan Montelius****.*

1 Introduction

Votre tâche est de simuler un algorithme d'ordonnancement. En partant de quelque chose qui ne fait presque rien nous allons aboutir à quelque chose qui est presque utile.

2 Tâches et files d'attente

Notre ordonnanceur gardera la trace d'un ensemble de tâches (*jobs*) qui seront dans l'une des trois listes : `ready`, `blocked` et `done`. Toutes les tâches seront créées au lancement du programme et placées dans la liste des tâches bloquées.

À chaque itération, le simulateur :

- débloque les tâches prêtes à être exécutées en les déplaçant de la liste `blocked` à la liste `ready`;
- planifie l'exécution d'une tâche de la liste `ready`;
- en fonction du résultat, la déplace vers l'une des trois listes.

Les tâches que le simulateur traitera ne feront bien sûr rien.

Le simulateur sait seulement de combien de temps une tâche a besoin pour aboutir et, au fur et à mesure que nous étendrons le simulateur, de combien de fois elle effectuera des opérations d'entrée/sortie (E/S).

Le simulateur collectera des statistiques et sera finalement capable de répondre à des questions telles que ; quel est le temps de traitement (*turnaround time*) et quel est le temps de réponse (*response time*) ?

Nous définissons une structure pour représenter les tâches et nous allons la garder simple pour commencer.

Les seules propriétés qui nous intéressent pour l'instant sont : l'instant d'arrivée (`arrival`), l'instant de déblocage (`unblock`), le temps d'exécution total (`exectime`) et le ratio des opérations d'E/S (`ioratio`).

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

L'instant de déblocage est initialement fixé à une valeur qui correspond au moment où la tâche arrive dans le système.

Si la tâche est bloquée à la suite d'une opération d'E/S, la valeur indiquera le moment où l'opération est terminée et où la tâche peut à nouveau être planifiée.

L'identifiant unique est quelque chose que nous créerons lorsque nous lancerons la simulation.

Pour mettre en place rapidement un *benchmark*, nous avons également créé une structure qui décrit les différentes tâches que nous allons utiliser.

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

L'initialisation pourrait alors ressembler à ceci : nous créons un tableau global avec 10 tâches et une spécification fictive à la fin. Les tâches peuvent être données dans n'importe quel ordre, bien qu'elles soient ici listées dans l'ordre de leur arrivée.

```
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
};
```

Nous avons trois files d'attente sous la forme de pointeurs globaux. Celles-ci seront implémentées sous la forme de listes chaînées de `jobs`. Ainsi, les files sont toutes des pointeurs nuls pour commencer.

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

La première chose que le simulateur doit faire est de parcourir les spécifications et de créer des tâches qui sont ajoutées à la file d'attente des tâches bloquées.

```
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++;
    }
}
```

Lorsque nous ajoutons les tâches à la file d'attente `blocked`, nous les ordonnons de manière à ce que les tâches ayant l'instant d'arrivée le plus bas soient les premières.

```
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;
}
```

Le simulateur garde la trace du temps et déplace les tâches de la file d'attente `blocked` vers la file d'attente `ready`. Comme les tâches sont ordonnées, il n'est pas nécessaire de parcourir toute la file d'attente. Un affichage à l'écran permet de retracer ce qui se passe.

```
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);
    }
}
```

Lors de la première exécution, nous ajoutons simplement des tâches à la fin de la file d'attente. Ce n'est peut-être pas optimal, mais pourquoi compliquer les choses ?

```
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;
}
```

Lorsque les tâches sont terminées, elles sont ajoutées à la liste des tâches terminées. L'ordre dans cette liste n'est pas important, alors autant les ajouter au début.

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

3 L'ordonnanceur

Le travail de l'ordonnanceur consiste à sélectionner la première tâche dans la file d'attente et à la laisser s'exécuter. Comme il ne s'agit que d'une simulation et que les tâches ne font rien, nous mettons simplement le temps d'exécution restant à zéro et la déplaçons dans la liste des tâches terminées.

Remarque : nous permettons actuellement aux tâches de s'exécuter jusqu'à ce qu'elles se terminent, il n'y a pas de préemption.

```
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;
    }
}
```

La procédure `schedule()` renvoie le temps qui s'est écoulé et pour cet ordonnanceur simple, il s'agit du temps d' exécution total de la tâche. S'il n'y a pas de tâche dans la file d'attente, nous retournons 1 pour faire avancer la simulation (pourquoi la file d'attente serait-elle vide ?).

La procédure n'a pas besoin pour l'instant de connaître l'heure, mais nous en aurons besoin plus tard et pour l'instant nous pouvons l'utiliser pour l'affichage.

Nous avons donc maintenant toutes les pièces du puzzle et nous pouvons créer notre premier ordonnanceur et le tester.

N'oubliez pas d'inclure les bons fichiers d'en-tête au début :

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
```

Par ailleurs, vous devez ordonner les procédures (ou les déclarer) afin de ne pas utiliser une procédure avant qu'elle ne soit déclarée.

```
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;
}
```

En espérant que tout cela fonctionne, ajoutons maintenant quelques fonctionnalités supplémentaires à l'ordonnanceur.

4 Shortest job first

La première chose à faire est d'adopter la stratégie de la tâche la plus courte (*shortest job first*). Tout sera identique, mais lorsque nous ajouterons des tâches à la file d'attente, nous les classerons de manière à ce que les tâches les plus courtes arrivent en premier.

Si vous ajoutez un test à la boucle while de `ready()`, vous devriez pouvoir le faire en un rien de temps. Le résultat n'est bien sûr pas immédiatement visible puisque le temps d'exécution total est le même. Ce qui aurait pu être amélioré est le temps de traitement moyen.

Pour vérifier si c'est le cas, ajoutons un champ à la structure `job`.

```
typedef struct job {
    // ...
    int turnaround;
    // ...
} job;
```

Dans la procédure `schedule()`, calculons maintenant le temps de traitement.

```
int schedule(int time) {
    // ...
    nxt->turnaround = time + exect - nxt->arrival;
    // ...
}
```

Dans la procédure principale, nous pouvons ensuite parcourir toutes les tâches dans la file d'attente des tâches effectuées et recueillir le temps de traitement moyen.

```c
int main() {
    // ...
    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);
    // ...
}
```

Essayez avec et sans trier la file d'attente `ready`, remarquez-vous une différence ?

5 Ordonnancement préemptif

Quel est le temps de réponse pour chacune des tâches ? Le temps de réponse est le temps écoulé entre l'arrivée et la première exécution.

Ajoutons un nouvel élément à la structure des tâches.

```c
typedef struct job {
    // ...
    int respons;
    // ...
} job;
```

Assurez-vous ensuite que ce champ est égal à zéro lorsque nous créons de nouvelles tâches dans la procédure `init()`. Mettez-le à zéro dans l'initialisation et lorsque la tâche commence à s'exécuter, calculez le temps de réponse. A la fin, calculez le temps de réponse moyen pour toutes les tâches.

Y a-t-il quelque chose que nous pouvons faire pour améliorer le temps de réponse ? Pourquoi ne pas adopter un ordonnanceur préemptif ? Fournissons un argument au simulateur qui définit l'intervalle de temps (*time slot*) que nous donnerons à chaque tâche. Passons ensuite ce paramètre à la procédure `schedule()`.

```c
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);
        // ...
    }
    // ...
}
```

Le temps d'exécution d'une tâche sera mis à jour pour tenir compte du temps de complétion. Chaque fois que nous ordonnançons une tâche, nous vérifions d'abord si le temps de complétion est inférieur ou égal à l'intervalle de temps donné. Si c'est le cas, le code est très similaire au précédent. Si, par contre, le temps de complétion est plus grand que l'intervalle de temps, nous devons décrémenter le temps et renvoyer le travail dans la file d'attente.

Nous utiliserons un zéro pour indiquer que le temps de réponse n'a pas encore été fixé (nous l'utiliserons plus tard).

Dans la fonction `schedule()`, nous pouvons maintenant vérifier si c'est la première fois que la tâche est ordonnancée, et si c'est le cas, nous pouvons entrer le temps de réponse comme l'heure actuelle moins l'heure d'arrivée.

Notez que nous devons maintenant vérifier si le temps de réponse a déjà été fixé. Si le temps de réponse est toujours zéro, nous savons que c'est la première fois que le travail est planifié et nous mettons à jour la valeur.

Voilà à quoi ça pourrait ressembler :

```
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;
```

Si vous essayez avec un intervalle de temps de 100, tout fonctionnera comme avant puisqu'aucune tâche n'a un temps d'exécution supérieur à 90. Si vous réduisez l'amplitude de l'intervalle de temps, vous verrez plus d'événements d' ordonnancement et vous pourriez voir le temps de réponse s'améliorer.

6 Entrées/sorties

Maintenant, pour le dernier problème, que se passe-t-il si une tâche effectue une opération d'entrée/sortie (E/S) ? Disons qu'une opération d'E/S prend 30 ms pour se terminer et que certaines des tâches effectueront une opération de temps en temps. Nous allons d'abord étendre notre simulation pour gérer les opérations d'E/S et ensuite essayer d'avoir un ordonnanceur plus intelligent.

Nous supposons que toutes les opérations d'E/S prennent 30 ms et définissons cette valeur dans une macro. Nous écrivons également une petite fonction qui tire à pile ou face et détermine si, compte tenu du rapport E/S de la tâche (`ioratio`), une opération d'E/S est effectuée. Si aucune opération d'E/S n'est effectuée, elle renvoie zéro, sinon elle renvoie le temps qui s'écoule avant que l'opération ne se produise (de `1` à `exect - 1`). Nous n'allons pas utiliser cette information maintenant, mais nous en aurons besoin plus tard.

```
#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;
}
```

Pour compiler le programme, vous devez maintenant inclure `math.h` et compiler avec la bibliothèque mathématique en utilisant l'option `-lm`.

Une fois que nous avons mis en place cette procédure, nous pouvons modifier la procédure `schedule()`. Notez que nous ne faisons rien d'autre pendant que la tâche effectue l'opération d'E/S. La procédure renverra simplement le temps d'exécution plus le temps qu'il a fallu pour effectuer l'opération d'entrée/sortie.

```
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;
```

Faites un essai et voyez ce qui se passe. Le temps d'exécution total va probablement augmenter et c'est tout à fait compréhensible, nous attendons la fin des opérations d'entrée/sortie. Y a-t-il quelque chose que nous puissions faire ?

7 Exécuter des tâches lors d'entrées/sorties

Un processeur ne fait rien pendant qu'une opération d'entrée/sortie est effectuée. Rester inactif et attendre est une perte de temps totale. Si nous pouvions ordonnancer une autre tâche pendant ce temps, nous pourrions améliorer la situation. Pourquoi ne pas renvoyer les tâches qui réalisent des E/S dans la file d'attente `blocked` jusqu'à ce qu'elles soient à nouveau prêtes à être exécutées ?

Il s'avère que nous avons toutes les pièces du puzzle pour le faire, il nous suffit de modifier légèrement l'ordonnanceur. Tout d'abord, nous fixons le temps d'exécution à la valeur renvoyée par la fonction `io_op()` (s'il est différent de zéro).

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

Ensuite, nous changeons ce que nous faisons si une opération d'E/S a été émise. Nous définissons la valeur `unblock` de la tâche et la renvoyons dans la file d'attente `blocked` au lieu de la file d'attente

`ready`. Nous modifions également l'affichage pour voir ce qui se passe.

```
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);
}
```

Que ce passe-t-il maintenant avec ce nouveau mécanisme implémenté ? Comment le temps d'exécution total évolue-t-il ?

8 Que faire de plus ?

L'ordonnanceur que nous avons maintenant est encore très simple. Il conserve toutes les tâches qui sont prêtes à être exécutées dans une file d'attente. La file d'attente est ordonnée de manière que les tâches qui ont un court temps de complétion soient traitées en premier. On peut cependant se demander si cela est toujours pertinent ou si cette information est connue. Si nous ne connaissons pas le temps de complétion, quelle tâche devons-nous sélectionner pour l'exécution ? Devrions-nous avoir des tâches avec des priorités différentes, devrions-nous avoir plusieurs files d'attente ? Si vous commencez à étendre cet ordonnanceur, vous aboutirez bientôt à quelque chose qui ressemble à un *multilevel feedback queue scheduler* où les tâches qui effectuent des opérations d'E/S sont prioritaires.