Université Joseph Fourier
GRENOBLE
UFR IM²AG

GINF41E0 - Second midterm
Year 2014

Answers hints

1 No more system bankrupt ?

Comment in a few lines the following sentences, the banker algorithm :

1. helps the system scheduler in its task of deciding which thread to schedule next.
   No, the banker algorithm is not related to the system scheduler. It is used to decide if an exclusive resource (thus, not the CPU) can be allocated without having the risk of a deadlock appearing in the system.
2. is used to restore a safe state in the system.
   No, it finds out if granting some allocation request will result in a safe state or not and deny it if not.
3. can detect deadlocks in a running system.
   No, this is only the safe state detection, that is a part of the banker algorithm, which can detect if the system is free of deadlock (but not the contrary).
4. is used for resources acounting in the system.
   No, it is only used to decide if a resource should be allocated or not depending on the risk of this allocation to cause a deadlock.
5. has a running time linear in the number of threads running in the system.
   No, its running time is quadratic in the number of threads (and linear in the number of resource kinds).
6. is executed at each clock interrupt.
   No, at each allocation request.
7. is necessary, otherwise the system eventually enters into deadlocks and ceases to function properly.
   No, the system might instead use a deadlock detection algorithm or might manage its resources in a way that avoids one of the necessary conditions for deadlocks.
8. is a part of most libraries that let users create and manage threads.
   No, its too costly to be included in that kind of libraries. Deadlocks in multithreaded programs are program flaws that should be debugged rather that prevented at runtime.

2 Painting session at the nursery school

Questions : All implementation you provide in your answer has to be in C using the POSIX API.

1. write the declaration of the shared data structures required to modelize the problem and write a function that will be called once to initialize these structures. Of course, you are expected to explain your code, comment it, or both.

```
#include <pthread.h>

// structure to modelize the state of a shared pot
struct shared_pot {
    int present, left_use, right_use;
};
```

```
struct shared_data {
    int C, B, n;
    int available_brushes;
    // state of shared pots (for each position and each color)
    struct shared_pot **shared_pots;
    int *available_pots;   // pots availability (for each color)

    // monitor lock
    pthread_mutex_t lock;
    // condition variables associated to brushes and pots (for each color)
    // a simpler solution would be to have only one condition variable
    pthread_cond_t brush_available;
    pthread_cond_t *pot_available;
};
```

```
#include <stdlib.h>
#include "Paint_and_Kids.h"

void initialize_shared_data(int C, int B, int *P, int n, struct shared_data *data) {
    int i,j;

    data->C = C;
    data->B = B;
    data->n = n;
    data->available_brushes = B;
    // Children are numbered from 1 to C, positions range from 0 (left of first child)
    // to C (right of last child)
    data->shared_pots = malloc(sizeof(struct shared_pot *)*(C+1));
    for (i=0; i<=C; i++) {
        // Colors are numbered from 1 to n, we don't use the index 0
        data->shared_pots[i] = malloc(sizeof(struct shared_pot)*(n+1));
        for (j=1; j<=n; j++) {
            data->shared_pots[i][j].present = 0;
            data->shared_pots[i][j].left_use = 0;
            data->shared_pots[i][j].right_use = 0;
        }
    }
    // Colors are numbered from 1 to n, we don't use the index 0
    data->available_pots = malloc(sizeof(int)*(n+1));
    for (j=1; j<=n; j++)
        data->available_pots[j] = P[j];

    pthread_mutex_init(&data->lock, NULL);
    pthread_cond_init(&data->brush_available, NULL);
    data->pot_available = malloc(sizeof(pthread_cond_t)*(n+1));
    for (j=1; j<=n; j++)
        pthread_cond_init(&data->pot_available[j], NULL);
}
```

2. write the functions associated to the children and to the teachers according to the previous specification. Once again, needless to say that you are expected to explain your code, comment it, or both.

```
#include <stdlib.h>
#include <stdio.h>
#include "Paint_and_Kids.h"
#include "Paint_and_Kids_threads.h"
```

```
int allocate_brush(struct shared_data *s, int number) {
    pthread_mutex_lock(&s->lock);
    while (!s->available_brushes) {
        printf("No brush available child %d started waiting\n", number);
        pthread_cond_wait(&s->brush_available, &s->lock);
    }
    s->available_brushes--;
    printf("Allocated one brush to child %d\n", number);
    pthread_mutex_unlock(&s->lock);
    return 1;
}

void free_brush(struct shared_data *s, int number) {
    pthread_mutex_lock(&s->lock);
    s->available_brushes++;
    pthread_cond_signal(&s->brush_available);
    printf("Child %d freed one brush\n", number);
    pthread_mutex_unlock(&s->lock);
}

int allocate_pot(struct shared_data *s, int number, int color) {
    int result;
    pthread_mutex_lock(&s->lock);
    // If a pot of the same color is already in use, do nothing
    if (!s->shared_pots[number-1][color].right_use &&
        !s->shared_pots[number][color].left_use) {
        // If a pot is already present, use it
        if (s->shared_pots[number-1][color].present) {
            s->shared_pots[number-1][color].right_use = 1;
            printf("Child %d started to use pot of %d paint on its left\n", number, color);
        } else if (s->shared_pots[number][color].present) {
            s->shared_pots[number][color].left_use = 1;
            printf("Child %d started to use pot of %d paint on its right\n", number, color);
        } else {
            // Otherwise allocate a new one and place it on the right
            // (this is an arbitrary choice)
            while (!s->available_pots[color]) {
                printf("No %d paint available child %d started waiting\n", color, number);
                pthread_cond_wait(&s->pot_available[color], &s->lock);
            }
            s->shared_pots[number][color].present = 1;
            s->shared_pots[number][color].left_use = 1;
            s->available_pots[color]--;
            printf("Allocated one pot of %d paint to child %d on its right\n", color, number);
        }
        result = 1;
    } else {
        printf("Ignored request from child %d who already has access to a pot of %d paint\n", r
        result = 0;
    }
    print_shared_state(s);
    pthread_mutex_unlock(&s->lock);
    return result;
}
```

```
void free_pot(struct shared_data *s, int number, int color) {
    struct shared_pot *pot = NULL;
    pthread_mutex_lock(&s->lock);
    if (s->shared_pots[number-1][color].right_use) {
        pot = &s->shared_pots[number-1][color];
        pot->right_use = 0;
        printf("Child %d has stoped using the pot of %d paint on its left\n", number, color);
    }
    if (s->shared_pots[number][color].left_use) {
        pot = &s->shared_pots[number][color];
        pot->left_use = 0;
        printf("Child %d has stoped using the pot of %d paint on its right\n", number, color);
    }
    if (!pot->left_use && !pot->right_use) {
        pot->present = 0;
        s->available_pots[color]++;
        pthread_cond_signal(&s->pot_available[color]);
        printf("A pot of %d paint has been freed\n", color);
    }
    print_shared_state(s);
    pthread_mutex_unlock(&s->lock);
}

void *child_behavior(void *arg) {
    int i;
    struct thread_data *t = (struct thread_data *) arg;
    struct shared_data *s = t->shared;

    int allocated_brushes = 0;
    int *allocated_pots = malloc(sizeof(int)*(s->n+1));
    for (i=0; i<=s->n; i++)
        allocated_pots[i] = 0;
    int total_allocated = 0;

    srandom((unsigned) (intptr_t) pthread_self());
    for (i=0; (i<t->iterations) || total_allocated; i++) {
        // allocate or free ?
        if ((i<t->iterations) && (random()%2)) {
            // brush or pot of paint ?
            if (random()%2) {
                // We avoid allocating several brushes to minimize deadlocks frequency
                if (!allocated_brushes) {
                    if (allocate_brush(s, t->number)) {
                        allocated_brushes++;
                        total_allocated++;
                    }
                }
            } else {
                int color = random()%s->n + 1;
                if (allocate_pot(s, t->number, color)) {
                    allocated_pots[color]++;
                    total_allocated++;
                }
            }
        } else {
            // brush or pot of paint ?
```

```
            if (random()%2) {
                if (allocated_brushes) {
                    free_brush(s, t->number);
                    allocated_brushes--;
                    total_allocated--;
                }
            } else {
                int color = random()%s->n + 1;
                if (allocated_pots[color]) {
                    free_pot(s, t->number, color);
                    allocated_pots[color]--;
                    total_allocated--;
                }
            }
        }
    }
    pthread_exit(NULL);
}
```

3. in this setup, deadlocks are possible, give an example of such a deadlock.
   An instance of the problem with 3 children, 2 colors and one pot of paint of each color is sufficient. There is a deadlock in the following situation :
   — *the first child has allocated a pot of the first color*
   — *the third child has allocated a pot of the second color*
   — *the first child wants a pot of the second color*
   — *the third child wants a pot of the first color*
4. do you think it is better to avoid or to detect deadlocks ? Explain why.
   In this situation detecting deadlock is useless because we do not have clue about how to break the deadlock (killing a child is not an option here). Thus, it's better to avoid them.
5. what algorithm presented during lectures seems appropriate to handle deadlocks in your simulation ? Explain why. If none seems appropriate, also explain why and explain how to adapt one of them.
   The banker algorithm is the closest to our needs, because we use resources that have several instances of each kind. To use this algorithm, we should adapt our simulation by having children anounce their maximal resources utilization. Furthermore, it is not designed to handle partially shared resources such as our pots of paint. By using the banker algorithm in this setup, we will detect unsafe states which could actually be free of deadlock because of this sharing capability. This comes in addition to the fact that the banker algorithm already overestimates the deadlock situations by ignoring the fact that threads might free some of their resources before allocating new ones.