CSE 306 Operating Systems

Concurrency: Mutual Exclusion and Synchronization

YoungMin Kwon

Concurrency

- Multiprogramming
  - Management of multiple processes within a uniprocessor system
- Multiprocessing
  - Management of multiple processes within multiprocessors
- Distributed processing
  - Management of multiple processes executing on multiple distributed computer systems

Concurrency

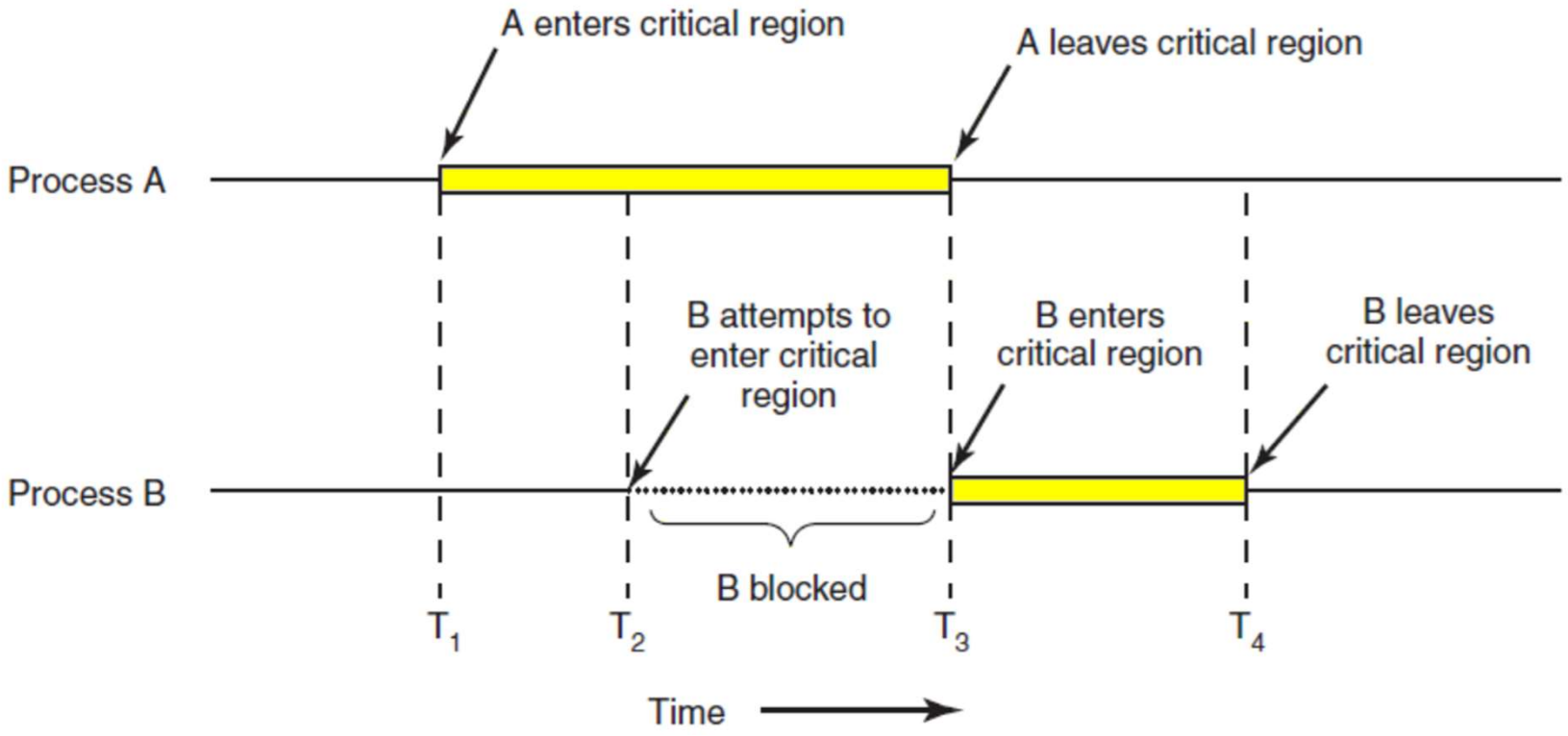
- Concurrency issues encompass:
  - Communication among processes
  - Sharing of resources, competing for resources
  - Synchronization of activities of processes
  - Allocation of processor time to processes

Terms Related to Concurrency

- Atomic operation
  - A sequence of instructions that appears to be indivisible
- Mutual exclusion
  - The requirement that when a process is using a shared resource, no other process may access the resource
- Critical section
  - A section of code that must NOT be executed while another process is in the corresponding section of the code

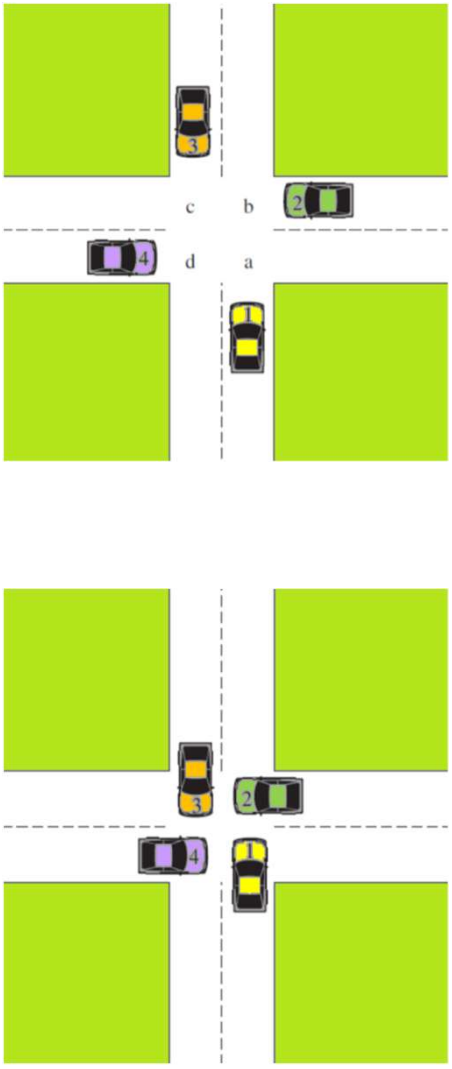
Terms Related to Concurrency

- Mutual exclusion using critical sections



Terms Related to Concurrency

- Deadlock
  - A situation where two or more processes are blocked while waiting for others to do something
- Livelock
  - A situation where two or more processes are changing their states without doing any useful work
- Starvation
  - A situation where a runnable process is overlooked indefinitely



Terms Related to Concurrency

- Race condition
  - The correctness of multiple threads or multiple processes depends on the timing of their execution

```
void *thread(void *vargp) {
    int myid = *((int*)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}

int main() {
    pthread_t tid[10];
    int i;

    for(i = 0; i < 10; i++)
        pthread_create(&tid[i], NULL, thread, &i);
    return 0;
}
```

Mutual Exclusion: Software Approaches

- Assumption
  - Simultaneous access (reading/writing) to the same location in memory are serialized

- Dekker's Algorithm
- Peterson's Algorithm

```c
// main.c
//
#define _GNU_SOURCE
#include <sched.h>
#include <pthread.h>
#include <stdio.h>
#include <time.h>

extern void enter_cs(long id);
extern void leave_cs(long id);

const long limit = 50000000;   //50 million
volatile long count = 0;

void *acc(void *vargp) {// thread function
    long id = (long)vargp;
    long i;
    for(i = 0; i < limit; i++) {
        enter_cs(id);   // enter critical section
        count++;        // shared resource
        leave_cs(id);   // leave critical section
    }
    return NULL;
}
```

```c
int main() {
    pthread_t tid1, tid2;
    clock_t start, diff;

    // schedule the process to core 0 (to use only 1 core)
    cpu_set_t set;
    CPU_ZERO(&set);    //set = {}
    CPU_SET(0, &set);  //set = {0}
    sched_setaffinity(0, sizeof(cpu_set_t), &set); //set of CPUs it can run

    start = clock();         // start time in usec

    pthread_create(&tid1, NULL, acc, (void*)0L);
    pthread_create(&tid2, NULL, acc, (void*)1L);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    diff = clock() - start; // elapsed time in usec

    printf("count: %ld\n", count);
    printf("elapsed time: %ld.%03ld msec\n", diff/1000, diff%1000);

    return 0;
}
```

```c
// dekker0.c
// No mutual exclusion
//

void enter_cs(long id) {
}

void leave_cs(long id) {
}
```

```
$ gcc main.c dekker0.c -lpthread

$ ./a.out
count : 88922583
elapsed time : 545.845 msec
```

Dekker's Algorithm 1st Attempt

```
/* PROCESS 0 */
while (turn != 0)
    ; /* do nothing */
/* critical section */
turn = 1;
```

```
/* PROCESS 1 */
while (turn != 1)
    ; /* do nothing */
/* critical section */
turn = 0;
```

- Mutual exclusion is guaranteed
- A process cannot enter CS consecutively
  - The pace of execution is dictated by the slower of the two processes
  - If one process fails, the other process is permanently blocked

```c
// dekker1.c
// Dekker's Algorithm (1st Attempt)
//
#include <stdio.h>

static volatile int turn = 0;

void enter_cs(long id) {
    while(turn != id)
        ;
    fprintf(stderr, "-");
}

void leave_cs(long id) {
    turn = (id == 1 ? 0: 1);
}
```

```
$ gcc main.c dekker1.c -lpthread

$ ./a.out
My biological life is too short to wait and see the result.
Change the limit to 5000 and check.
```

Dekker's Algorithm 2nd Attempt

```
/* PROCESS 0 */
while (flag[1])
    ; /* do nothing */
flag[0] = true;
/* critical section */
flag[0] = false;
```

```
/* PROCESS 1 */
while (flag[0])
    ; /* do nothing */
flag[1] = true;
/* critical section */
flag[1] = false;
```

- Mutual exclusion is NOT guaranteed
  - P0 executes the while statement and finds that flag[1] is false
  - P1 executes the while statement and finds that flag[0] is false
  - P0 sets flag[0] to true and enters the critical section
  - P1 sets flag[1] to true and enters the critical section

```c
// dekker2.c
// Dekker's Algorithm (2nd Attempt)
//

static volatile int flag[2] = {0, 0};

void enter_cs(long id) {
    long other = (id == 1 ? 0: 1);

    while(flag[other])
        ;
    flag[id] = 1;
}

void leave_cs(long id) {
    flag[id] = 0;
}
```

```
$ gcc main.c dekker2.c -lpthread

$ ./a.out
count: 99999998
elapsed time: 1003.850 msec
```

Dekker's Algorithm 3rd Attempt

```
/* PROCESS 0 */
flag[0] = true;
while (flag[1])
    ; /* do nothing */
/* critical section */
flag[0] = false;
```

```
/* PROCESS 1 */
flag[1] = true;
while (flag[0])
    ; /* do nothing */
/* critical section */
flag[1] = false;
```

- Mutual exclusion is guaranteed
- Deadlock can happen
  - P0 sets flag[0] to true
  - P1 sets flag[1] to true
  - P0 waits in the while statement until flag[1] is cleared
  - P1 waits in the while statement until flag[0] is cleared

```c
// dekker3.c
// Dekker's Algorithm (3rd Attempt)
//
#include <stdio.h>

static volatile int flag[2] = {0, 0};

void enter_cs(long id) {
    long other = id ? 0: 1;

    flag[id] = 1;
    while(flag[other])
        ;
    fprintf(stderr, "-"); //to see if threads are deadlocked
}

void leave_cs(long id) {
    flag[id] = 0;
}
```

```
$ gcc main.c dekker3.c -lpthread

$ ./a.out
You will see a deadlock
```

Dekker's Algorithm 4th Attempt

```
/* PROCESS 0 */
flag[0] = true;
while (flag[1]) {
    flag[0] = false;
    /* delay */
    flag[0] = true
}
/* critical section */
flag[0] = false;
```

```
/* PROCESS 1 */
flag[1] = true;
while (flag[0]) {
    flag[1] = false;
    /* delay */
    flag[1] = true
}
/* critical section */
flag[1] = false;
```

- Mutual exclusion is guaranteed
- Livelock is possible
  - P0 sets flag[0] to true
  - P1 sets flag[1] to true
  - P0 checks flag[1]
  - P1 checks flag[0]
  - P0 sets flag[0] to false
  - P1 sets flag[1] to false
  - P0 sets flag[0] to true
  - P1 sets flag[1] to true
  - …

```
// dekker4.c
// Dekker's Algorithm (4th Attempt)
//
static volatile int flag[2] = {0, 0};

void enter_cs(long id) {
    long other = id ? 0: 1;

    flag[id] = 1;      //*** Possible livelock ***
    while(flag[other]) {
        flag[id] = 0;
        {int i = 0; while(i++ < 100) ;} //delay
        flag[id] = 1;
    }
}

void leave_cs(long id) {
    flag[id] = 0;
}
```

```
$ gcc main.c dekker4.c -lpthread

$ ./a.out
count: 100000000
elapsed time: 1207.643 msec
```

Dekker's Algorithm

```
/* PROCESS 0 */
flag[0] = true;
while (flag[1]) {
    if (turn == 1) {
        flag[0] = false;
        while(turn == 1)
            ; /* do nothing*/
        flag[0] = true
    }
}
/* critical section */
turn = 1;
flag[0] = false;
```

```
/* PROCESS 1 */
flag[1] = true;
while (flag[0]) {
    if (turn == 0) {
        flag[1] = false;
        while(turn == 0)
            ; /* do nothing*/
        flag[1] = true
    }
}
/* critical section */
turn = 0;
flag[1] = false;
```

- Break the mutual courtesy
  - turn variable indicates which process has the right to insist

```c
// dekker_final.c
// Dekker's Algorithm (final version)
//
static volatile int turn = 0;
static volatile int flag[2] = {0, 0};

void enter_cs(long id) {
    long other = (id ==1 ? 0: 1);

    flag[id] = 1;
    while(flag[other]) {
        if(turn == other) {
            flag[id] = 0;
            while(turn == other)
                ;
            flag[id] = 1;
        }
    }
}

void leave_cs(long id) {
    turn = (id == 1 ? 0: 1);
    flag[id] = 0;
}
```

```
$ gcc main.c dekker_final.c -lpthread

$ ./a.out
count: 100000000
elapsed time: 2009.403 msec
```

Peterson's Algorithm

```
/* PROCESS 0 */
flag[0] = true;
turn = 1;
while (flag[1] && turn == 1)
    ; /* do nothing */
/* critical section */
flag[0] = false;
```

```
/* PROCESS 1 */
flag[1] = true;
turn = 0;
while (flag[0] && turn == 0)
    ; /* do nothing */
/* critical section */
flag[1] = false;
```

- Break the deadlock in the 2nd attempt
  - turn is either 0 or 1

```c
// peterson.c
// Peterson's Algorithm
//
#include <stdio.h>
static volatile int turn = 0;
static volatile int flag[2] = {0, 0};

void enter_cs(long id) {
    long other = (id == 1 ? 0: 1);

    flag[id] = 1;
    turn = other;
    while(flag[other] && (turn == other))
        ;
    fprintf(stderr, "-");
}

void leave_cs(long id) {
    flag[id] = 0;
}
```

```
$ gcc main.c peterson.c -lpthread

$ ./a.out
Again, it takes long time to finish.
Change the limit to 5000 and check.
Why is it so slow?
```

Mutual Exclusion: HW support

- Disabling interrupts
  - Works with a uniprocessor system
  - A process will continue to run until
    - It makes a system call
    - An interrupt occurs

```
void enter_cs(int id /*unused*/) {
    // disable interrupts
}

void leave_cs(int id /*unused*/) {
    // enable interrupts
}
```

Mutual Exclusion: HW support

- Atomic instructions
  - Carry out two or more actions atomically
  - Instructions are treated as a single step that cannot be interrupted

- Atomic compare and swap instruction

```
int compare_and_swap(int *word, int testval, int newval) {
    int oldval;
    oldval = *word;
    if(oldval == testval)
        *word = newval;
    return oldval;
}
```

```
// compare_and_swap.c
// Using __sync_val_compare_and_swap
//

static volatile int token = 0;

void enter_cs(int id /*unused*/) {
    while(__sync_val_compare_and_swap( &token /*ptr*/
                                     , 0      /*old val*/
                                     , 1      /*new val*/ ) == 1)
        ;
}

void leave_cs(int id /*unused*/) {
    token = 0;
}
```

```
$ gcc main.c compare_and_swap.c -lpthread

$ ./a.out
count: 100000000
elapsed time: 2875.683 msec
```

Mutual Exclusion: HW support

- Atomic exchange instruction

```
void exchange(int *reg, int *mem)
{
    int tmp;
    tmp = *mem;
    *mem = *reg;
    *reg = tmp;
}
```

```
static volatile int token = 0;

void enter_cs(int id) {
    int key = 1;
    do exchange(&key, &token)
    while(key != 0);
}

void leave_cs(int id) {
    token = 0;
}
```

Mutual Exclusion: HW support

- Advantages of the atomic instructions
  - Applicable to any number of processes on either a single processor or multiple processors
  - Simple and easy
  - Can support multiple critical sections

- Disadvantages of the atomic instructions
  - Busy waiting is employed
  - Starvation is possible
  - Deadlock is possible