6

CS2106 Tutorial Group 03/04

She Jiayu

Recap

Synchronization

- Race conditions
  - Unsynchronized access to shared modifiable resources
- Critical section
- Mutual exclusion algorithm
  - Mutual exclusion
  - Progress
  - No starvation

Synchronization

- Synchronization techniques
  - Software algorithm
    - Busy wait
    - Peterson's
    - Bakery
  - Software abstractions
    - Mutex/Semaphore
    - Monitor
  - Hardware
    - Disable interrupts
    - Machine level instructions (TestAndSet)

Tutorial 06

Q1

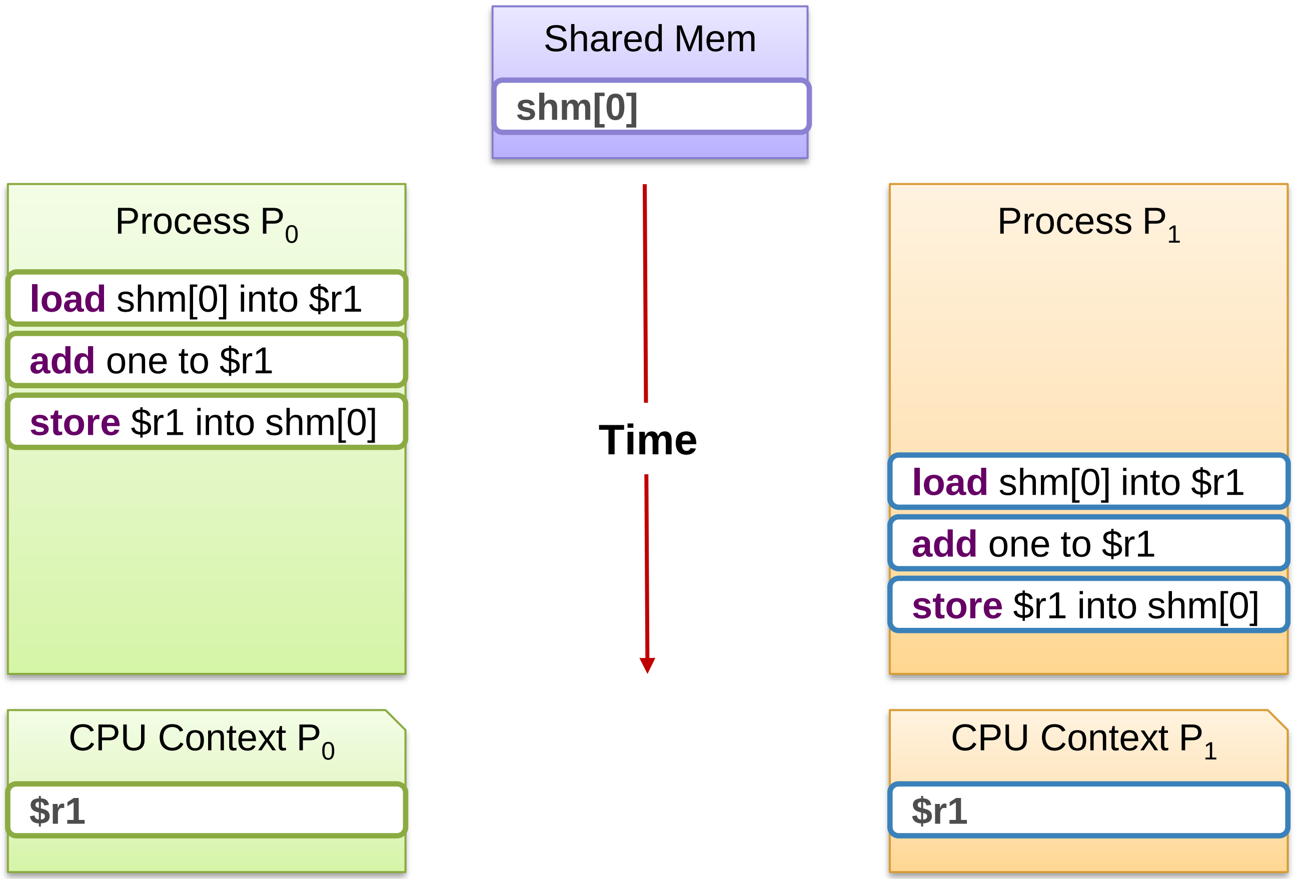
- Multi-core platform X does not support semaphores or mutexes. However, platform X supports the following atomic function:
  - `bool _sync_bool_compare_and_swap (int* t, int v, int n);`
- The above function atomically compares the value at location pointed by t with value v. If equal, the function will replace the content of the location with a new value n, and return 1 (true), otherwise return 0 (false).

Q1

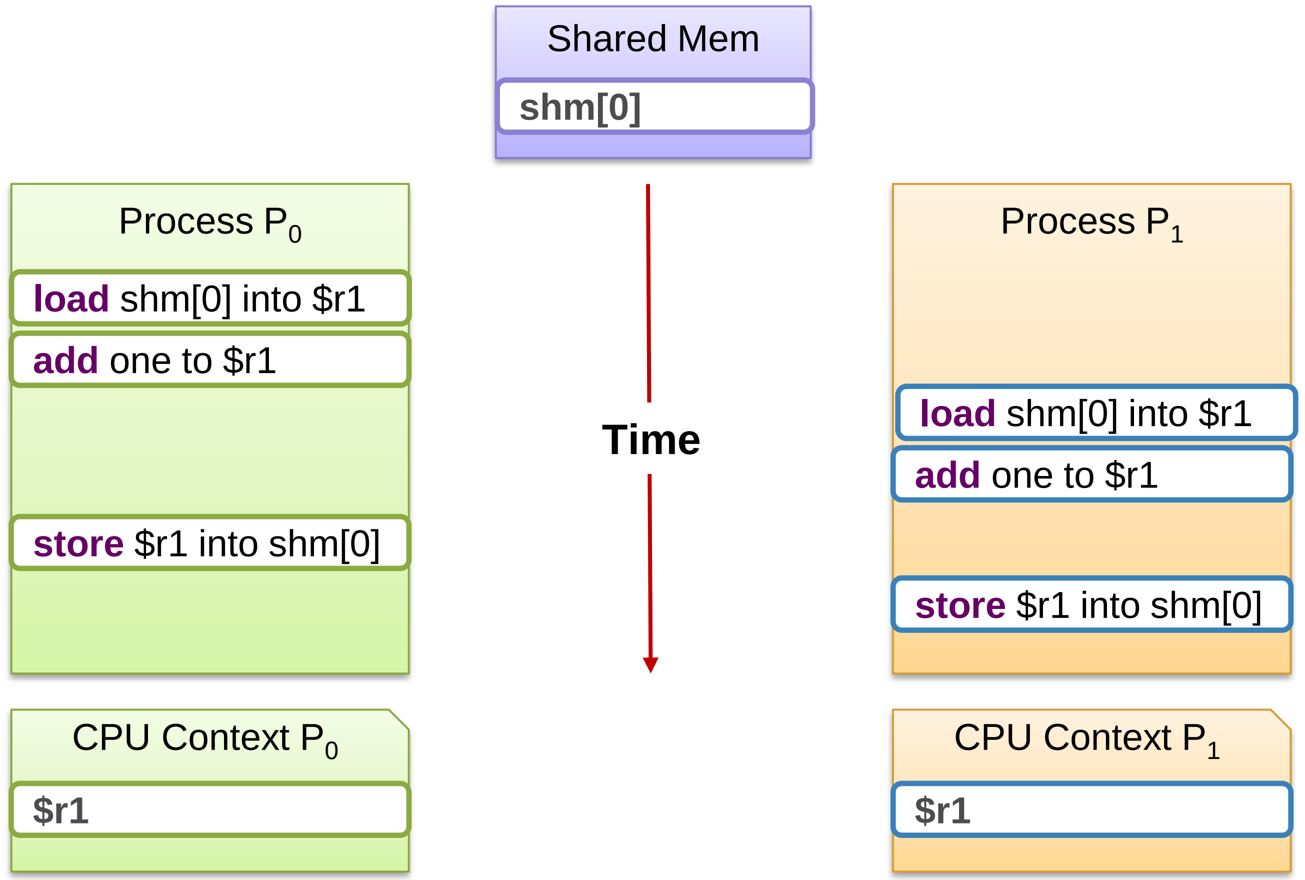
- Your task is to implement function **`atomic_increment`** on platform X. Your function should always return the incremented value of referenced location t, and be free of race conditions. The use of busy waiting is allowed.

```
int atomic_increment( int* t ) {
    //your code here
}
```

Q1


Shared Mem
shm[0]
Process P_0
load shm[0] into $r1
add one to $r1
store $r1 into shm[0]
Time
Process P_1
load shm[0] into $r1
add one to $r1
store $r1 into shm[0]
CPU Context P_0
$r1
CPU Context P_1
$r1

Q1


Shared Mem
shm[0]
Process P_0
load shm[0] into $r1
add one to $r1
store $r1 into shm[0]
Time
Process P_1
load shm[0] into $r1
add one to $r1
store $r1 into shm[0]
CPU Context P_0
$r1
CPU Context P_1
$r1

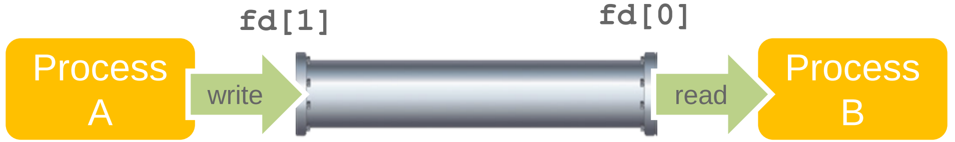
Q1

```
int atomic_increment( int* t ) {
    int temp;
    do {
        temp = *t;
    } while (!_sync_bool_compare_and_swap(t, temp, temp+1));
    return temp+1;
}
```

Q2

- Implement an intra-process mutual exclusion mechanism using Unix pipes, without using mutex (pthread_mutex), semaphore (sem), or any other synchronization construct.
- You are given the pipe-based lock (**`struct pipelock`**). You need to write code for **`lock_init`**, **`lock_acquire`**, and **`lock_release`**.

Q2



- In multithreaded processes, file descriptors are shared between all threads in a process. If multiple threads simultaneously call **`read()`** on a file descriptor, only one thread will be successful in reading any available data up to the buffer size provided, the others will remain blocked until more data is available to be read. The read end of a pipe is at index 0, the write end at index 1.

```
int pipe(int pipefd[2]);
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Q2

```
void init(Semaphore sem) {
    sem = new Semaphore(1);
}

void acquire(Semaphore sem) {
    wait(sem);
}

void release(Semaphore sem) {
    signal(sem);
}
```

```
struct pipelock {
    int fd[2];
};

void lock_init(struct pipelock *lock) {
    pipe(lock->fd);
    write(lock->fd[1], "a", 1);
}
void lock_acquire(struct pipelock *lock) {
    char c;
    read(lock->fd[0], &c, 1);
}
void lock_release(struct pipelock * lock) {
    write(lock->fd[1], "a", 1);
}
```

Q3

- We examine the stubborn villagers problem. A village has a long but narrow bridge that does not allow people crossing in opposite directions to pass by each other. All villagers are very stubborn, and will refuse to back off if they meet another person on the bridge coming from the opposite direction.

Q3a

- Explain how the behavior of the villagers can lead to a deadlock.
- Two villagers on different sides of the bridge trying to cross at the same time will lead to a deadlock.

Q3b

- Analyse the correctness of the following solution and identify the problems, if any.

```
Semaphore sem = 1;

void enter_bridge() {
    sem.wait();
}

void exit_bridge() {
    sem.signal();
}
```

Q3b

- The problem with this solution is that it allows only a single villager to cross at a time. A second villager crossing the bridge in the same direction cannot walk behind the first one and instead needs to wait for the first one to exit the bridge.

Q3c

- Modify the above solution to support multiple people crossing the bridge in the same direction. You are allowed to use a single shared variable and a single semaphore.

Q3c

```
Semaphore mutex=1; int crossing = 0;
void enter_bridge_direction1() {
    bool pass=false;
    while(!pass) {
        mutex.wait();
        if(crossing>=0) {
            crossing++;
            pass=true;
        }
        mutex.signal();
    }
}
void enter_bridge_direction2() {
    bool pass=false;
    while(!pass) {
        mutex.wait();
        if(crossing<=0) {
            crossing--;
            pass=true;
        }
        mutex.signal();
    }
}
void exit_bridge_direction1() {
    mutex.wait();
    crossing--;
    mutex.signal();
}
void exit_bridge_direction2() {
    mutex.wait();
    crossing++;
    mutex.signal();
}
```

Q3d

- What is the problem with solution in (c)?

- The problem with this solution is that it allows the villagers crossing in one direction to indefinitely starve the villagers crossing in the other direction.

Q4

- We mentioned that general semaphore (S > 1) can be implemented by using binary semaphore (S == 0 or 1). Consider the following attempt:

Q4

```
int count = N;          //any non-negativeinteger
Semaphore mutex = 1;    //binary semaphore
Semaphore queue = 0;    //binary semaphore, for blocking tasks

GeneralWait() {
    wait( mutex );
    count = count - 1;
    if (count < 0) {
        signal( mutex );
        wait( queue );
    } else {
        signal( mutex );
    }
}

GeneralSignal() {
    wait( mutex );
    count = count + 1;
    if (count <= 0) {
        signal( queue );
    }
    signal( mutex );
}
```

Q4a

- The solution is very close, but unfortunately can still have undefined behavior in some execution scenarios. Give one such execution scenario to illustrate the issue.
  - Hint: binary semaphore works only when its value S = 0 or S = 1.

Q4a

- The issue is task interleaving between **`signal(mutex)`** and **`wait(queue)`** in **`GeneralWait()`** function. Consider the scenario where count is 0, two tasks A and B execute **`GeneralWait()`**, as task A clears the **`signal(mutex)`**, task B gets to executes until the same line. At this point, count is -2. Suppose two other tasks C and D now executes **`GeneralSignal()`** in turns, both of them will perform **`signal(queue)`** due to the count -2. Since queue is a binary semaphore, the second **`signal(queue)`** will have undefined behavior (remember that we cannot have S = 2 for binary semaphore).

Q4b

- Correct the attempt. Note that you only need very small changes to the two functions.

Q4b

```
int count = N;
Semaphore mutex = 1;
Semaphore queue = 0;

GeneralWait() {
    wait( mutex );
    count = count - 1;
    if (count < 0) {
        signal( mutex );
        wait( queue );
    } else {
        signal( mutex );
    }
}

GeneralSignal() {
    wait( mutex );
    count = count + 1;
    if (count <= 0) {
        signal( queue );
    }
    signal( mutex );
}
```

```
int count = N;
Semaphore mutex = 1;
Semaphore queue = 0;

GeneralWait() {
    wait( mutex );
    count = count - 1;
    if (count < 0) {
        signal( mutex );
        wait( queue );
    }   //else removed
    signal( mutex );
}

GeneralSignal() {
    wait( mutex );
    count = count + 1;
    if (count <= 0) {
        signal( queue );
    } else {   //else added
        signal( mutex );
    }
}
```

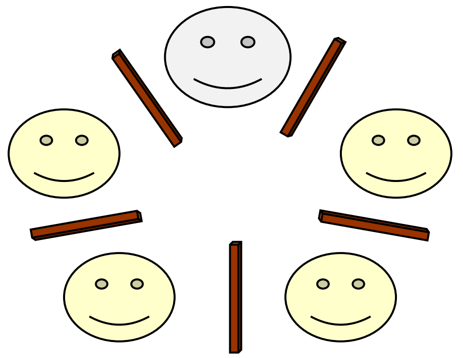
Q4b

- Using the same execution scenario in (a), task D will not be able to do the 2nd **`signal(queue)`** as the mutex is not unlocked. Either Task A or B can clears the **`wait(queue)`**, then **`signal(mutex)`** allowing task D to proceed. At this point in time, the queue value has settled back to 0, so there is no undefined **`signal(queue)`**.
- Reference
  - *D. Hemmendinger*, "*A correct implementation of general semaphores*", Operating Systems Review, vol. 22, no. 3 (July, 1988), pp. 42-44.

Q5

- Our philosophers in the lecture are all left-handed (they pick up the left chopstick first). If we force one of them to be a right-hander, i.e. pick up the right chopstick before the left, then it is claimed that the philosophers can eat without explicit synchronization.
- Do you think this is a deadlock free solution to the dining philosopher problem?

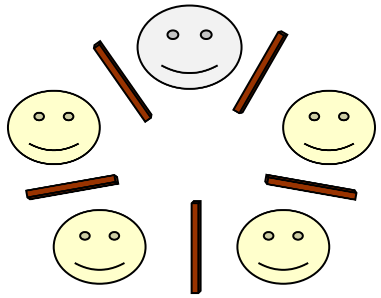
Q5



```
while (TRUE){
    Think( );
    //hungry, need food!
    takeChpStck( LEFT );
    takeChpStck( RIGHT );

    Eat( );

    putChpStck( LEFT );
    putChpStck( RIGHT );
}
```

```
while (TRUE){
    Think( );
    //hungry, need food!
    takeChpStck( RIGHT );
    takeChpStck( LEFT );

    Eat( );

    putChpStck( RIGHT );
    putChpStck( LEFT );
}
```

Q5



- The claim is true.
- For ease of discussion, let's refer to the right-hander as R.
  - If R grabbed the right fork then managed to grab the left fork, then R can eat, not a deadlock.
  - If R grabbed the right fork but the left fork is taken, then the left neighbor of R has already gotten both forks and been eating, eventually release fork.
  - If R cannot grabbed the right fork, then the right neighbor of R has taken its left fork. In the worst case, all remaining left-hander all hold on to their left fork. The left neighbor of R will be able to take its right fork because R is still trying to get its right fork.

Thanks!

Any questions?