CS302
Operating System
Lab 4

Concurrency: Mutual Exclusion and Synchronization

March 28th , 2018

Dongping Zhang
cadongllas@gmail.com

Race Condition

- The outcome of an execution depends on a particular order in which the shared resource is accessed.
- A simple example
  - a.c and b.c are two processes need to display their outputs on the standard error
- Compile and run like this:

  ```
  gcc a.c -o a
  gcc b.c -o b
  ./a & ./b &
  ```

Mutual Exclusion

- Mutual exclusion
  - ➢It prevents multiple threads from entering
- Critical resource:
  - ➢Nonsharable resource
  - ➢Example: only one process at a time is allowed to send command to the printer
- critical section
  - ➢the portion of the program that uses critical resource
  - ➢Only one program at a time is allowed in its critical section
- Lock
  - ➢a mechanism for mutual exclusion

Semaphores

- two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal.

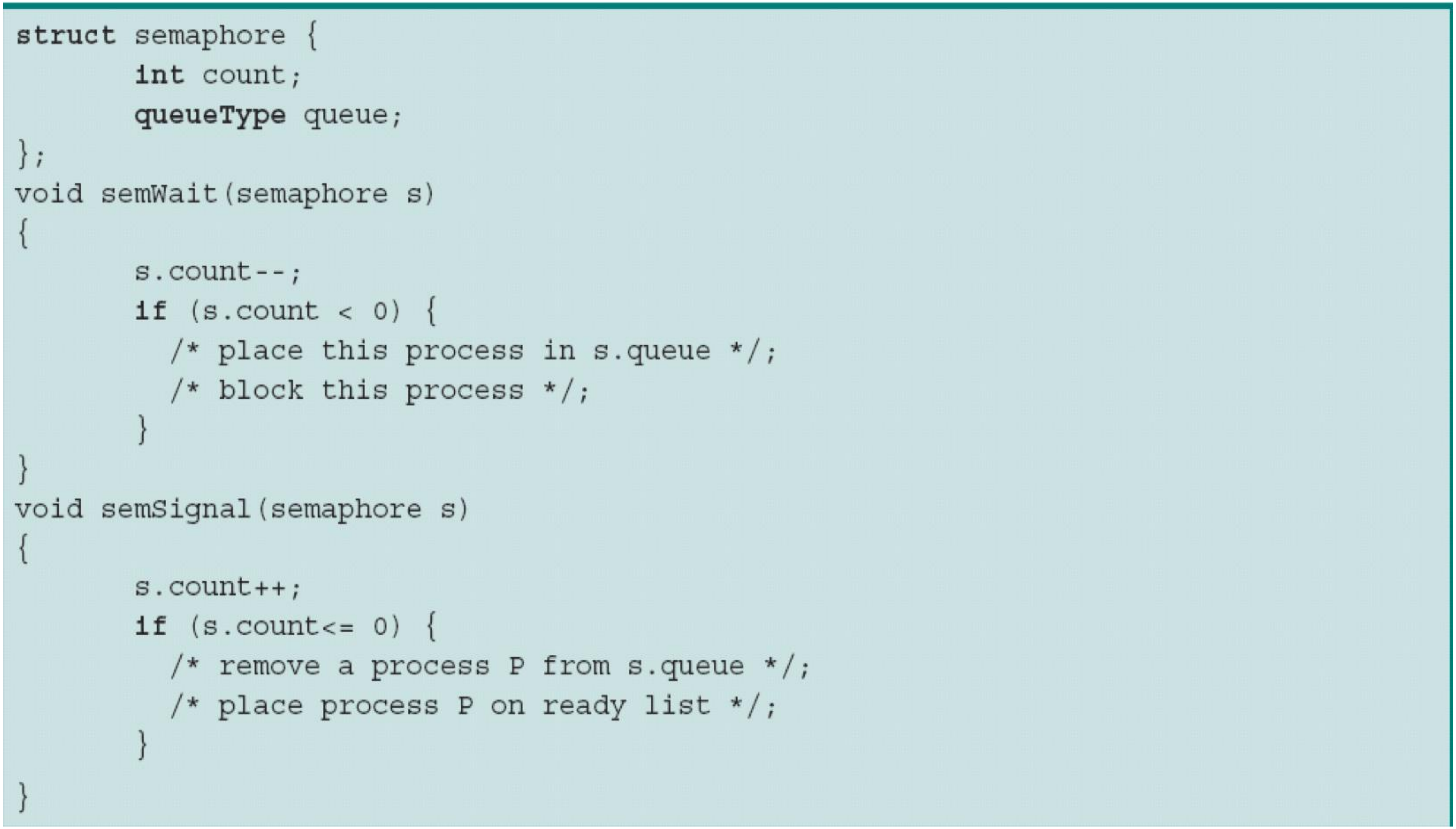
- For signaling, special variables called **semaphores** are used.

- If a process is waiting for a signal, it is suspended until that signal is sent

Semaphores

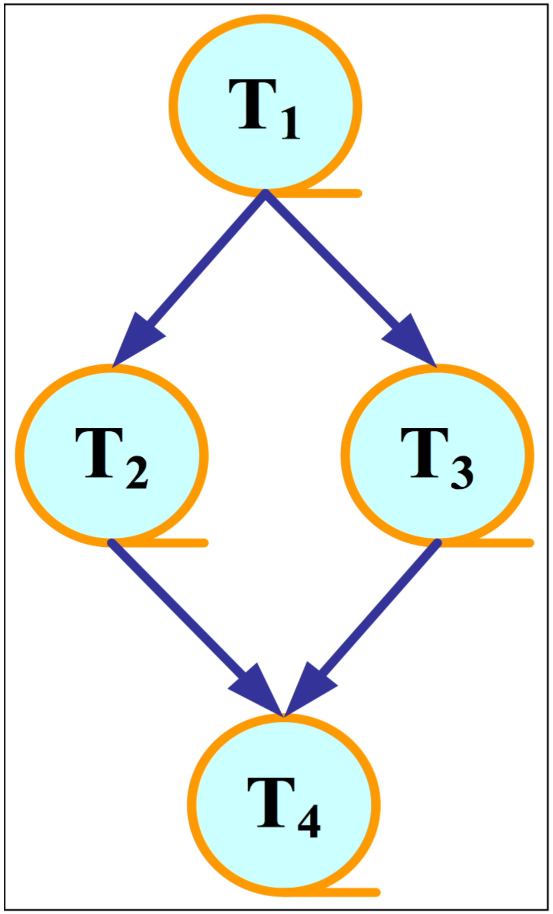
- Semaphore is a variable that has an integer value
  - ➢Initialize: a nonnegative integer value
  - ➢semWait (P): decreases the semaphore value. the value becomes negative, then the process executing the semWait is blocked.
  - ➢semSignal (V): increases semaphore value. If the resulting value is less than or equal to zero, then a process is blocked by a semWait operation, if any, is unblocked.

Semaphores



```
struct semaphore {
       int count;
       queueType queue;
};
void semWait(semaphore s)
{
       s.count--;
       if (s.count < 0) {
         /* place this process in s.queue */;
         /* block this process */;
       }
}
void semSignal(semaphore s)
{
       s.count++;
       if (s.count<= 0) {
         /* remove a process P from s.queue */;
         /* place process P on ready list */;
       }
}
```

Semaphores



b1,b2,b3:semaphore : = 0,0,0

T1: { … V(b1); V(b1); }

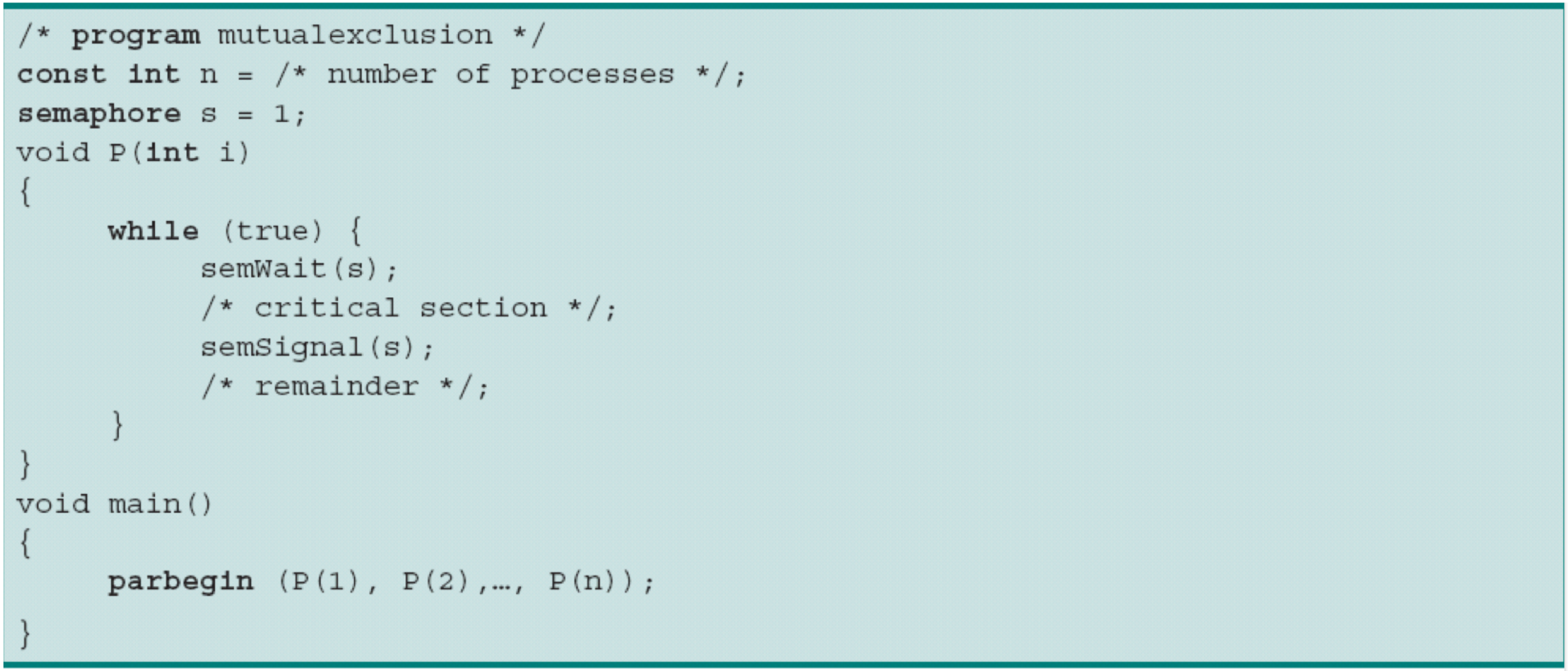
T2: { P(b1); … V(b2); }

T3: { P(b1); … V(b3); }

T4: { P(b2); P(b3); … }

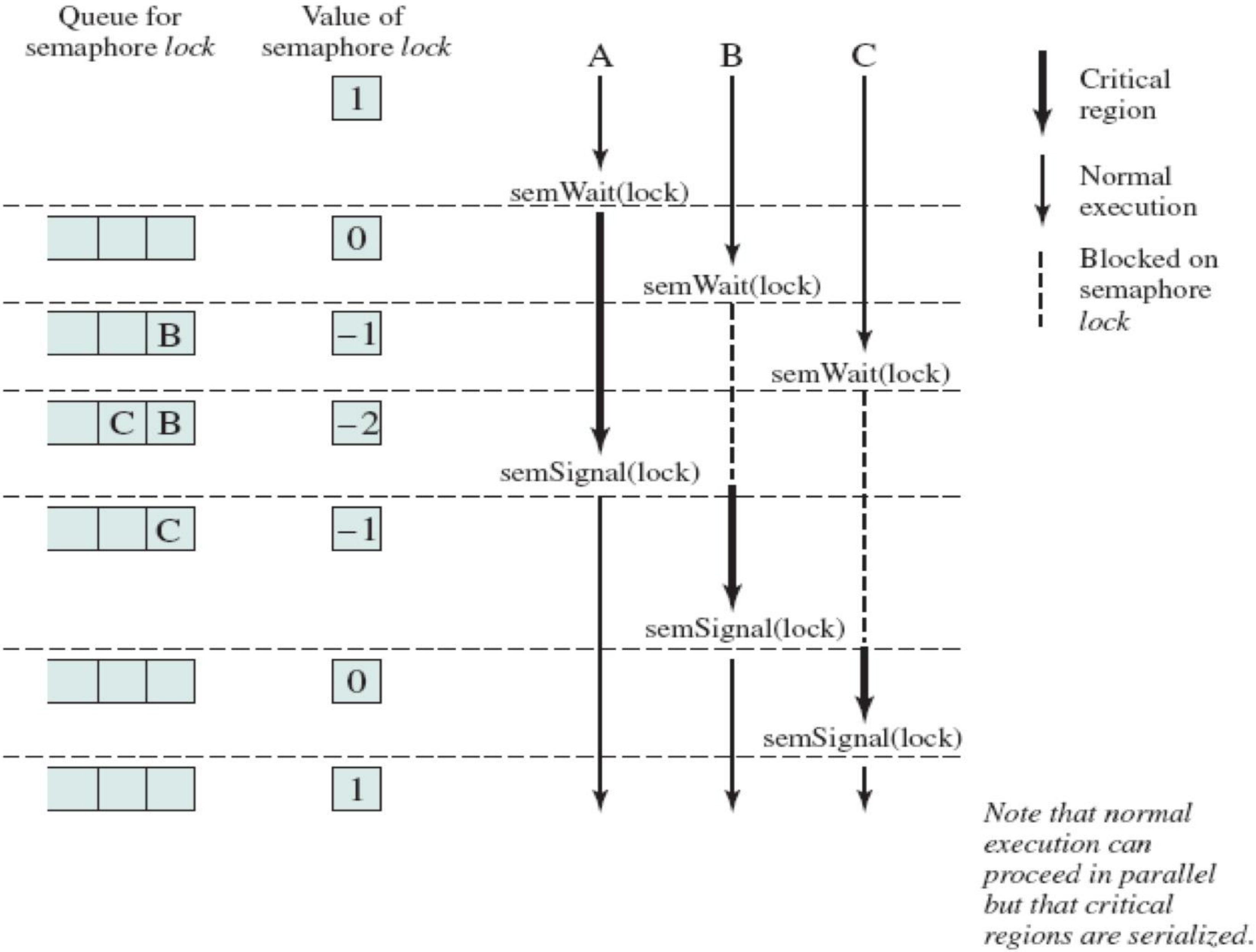
（因在T2和T3中分别对b2、b3做了V操作，所以T4要用两个P操作）

Mutual Exclusion using Semaphores



```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section */;
          semSignal(s);
          /* remainder */;
     }
}
void main()
{
     parbegin (P(1), P(2),…, P(n));
}
```

Mutual Exclusion using Semaphores


Queue for semaphore *lock*
Value of semaphore *lock*
A
B
C
1
semWait(lock)
0
semWait(lock)
B
–1
semWait(lock)
C
B
–2
semSignal(lock)
C
–1
semSignal(lock)
0
semSignal(lock)
1
Critical region
Normal execution
Blocked on semaphore *lock*
Note that normal execution can proceed in parallel but that critical regions are serialized.

Semaphore in C

- semaphore.c shows how to use these functions to create, operate and remove named semaphore.
- compile semaphore.c like this: gcc semaphore.c -pthreaad -o semaphor

Function	Description
sem_open	Opens/creates a named semaphore for use by a process
sem_wait	lock a semaphore
sem_post	unlock a semaphore
sem_close	Deallocates the specified named semaphore
sem_unlink	Removes a specified named semaphore

Shared Output: Use semaphore

- We use semaphore to provide mutual exclusion to the standard error. If the process is using, the another process will wait until the semaphore is unlocked.
- Compile and run：

```
gcc a_sol.c -pthread -o a
gcc b_sol.c -pthread -o b
./a & ./b &
```