Chapter 6 & 7: Process Synchronization

Process Synchronization

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
  - Concurrent access to shared data may result in data inconsistency
- An example
  - The variable counter in an implementation of producer-consumer

```
item  next_produced;
while (true) {
      /* produce an item in next_produced */
      while (counter == BUFFER_SIZE) //buffer is full
               ;/* do nothing, such as create data */
      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

```
item  next_consumed;
while (true) {
     while (counter == 0) //buffer is empty
              ; /* do nothing */
     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
      counter--;  //total # of items in the buffer
     /* consume the item in next_consumed */
}
```

Race Condition

- counter++ could be implemented as
  register1 = counter
  register1 = register1 + 1
  counter = register1
- counter-- could be implemented as
  register2 = counter
  register2 = register2 - 1
  counter = register2
- Consider this execution interleaving with "counter = 5" initially:
  S0: producer execute register1 = counter {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = counter {register2 = 5}
  S3: consumer execute register2 = register2 – 1 {register2 = 4}
  S4: producer execute counter = register1 {counter = 6 }
  S5: consumer execute counter = register2 {counter = 4}

Race condition:

- several processes access and manipulate the same data concurrently
- outcome depends on which order each access takes place.

Critical Section Problem

- Consider system of *n* processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Several processes may be changing common variables, updating table, writing file, etc.
  - critical section
    - The segment of code in a process that modifies these shared variables, tables, files
- When one process is in critical section, other process should not enter their critical sections for these shared data .

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Critical Section

- General structure of process P_i

```
do {
        entry section          Ask for permission

            critical section   Protect this section

        exit section           Do sth in order to allow other
                               processes to enter critical section

            remainder section

} while (true);
```

Critical Section Problem

- Critical section problem is to design a protocol（协议）that the processes can use to cooperate
  - Process with critical section should follow the following steps
    1. execute entry section to ask for permission
    2. then execute critical section,
    3. execute exit section to allow other process to enter critical section
    4. then execute remainder section

Solution to Critical-Section Problem

- A solution to the critical-section problem must satisfy three requirements:
  1. Mutual Exclusion
     - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
  2. Progress
     - If no process is executing in its critical section and if other processes want to enter critical section, one of them must be selected. They cannot be postponed indefinitely
  3. Bounded Waiting
     - If a process has made a request to enter its critical section, then, before that request is granted, there is a bound to the times that others can enter critical section

Solution to Critical-Section Problem

- Assumptions in the solution
  - each process executes at a nonzero speed
  - no concerning relative speed of the ***n*** processes
  - the **load** and **store** machine-language instructions are atomic (that is, cannot be interrupted)

Example: Solution to Critical-Section Problem

Example: two students want to eat from the same plate of food using one spoon.

1. **Mutual Exclusion** - If one student is eating using the spoon, then the other student must wait for the spoon to become available: the two students cannot both use the spoon at the same time! Otherwise the two students will end up fighting and breaking the plate of food (race condition).
2. **Progress** - If one student is not hungry and the other student wants to get the spoon to eat, then the other student must be able to take the spoon and eat: the student who is not hungry is not allowed to prevent the other student from taking the spoon and eating. Otherwise the other student will wait for ever and die from hunger.
3. **Bounded Waiting** - If one student is eating and the other student is waiting for the spoon, the first student is not allowed to put down the spoon on the table and immediately take it again to eat again: the student who puts down the spoon must allow the other student to take the spoon so that the other student can eat too. In other words, the two students must be nice with each other and not be selfish. Otherwise the other student will wait for ever and die from hunger.

Critical Section: Solution 1

turn's value is initialized to be either ***i*** or ***j***

P_i	P_j

do {
    while (turn == j); //entry section
        critical section //turn is i
    turn = j; //exit section
        remainder section
} while (true);
	do {
	    while (turn == i); //entry section
	        critical section // turn is j
	    turn =i; //exit section
	        remainder section
	} while (true);

Mutual Exclusion : Yes

Progress : No

Bounded waiting : Yes

Detailed Explanation on Example 1

1. Mutual Exclusion: Yes - If process P_i is inside the critical section then P_j cannot enter the critical section because of the **`while`** loop in the entry section that forces P_j to wait for P_i to exit the critical section.
    1. And vice-versa: if P_j is inside the critical section then P_i must wait in the entry section until P_j exits the critical section.
    2. What happens if both processes try to enter the critical section at the exact same time? The shared **`turn`** variable can only store **`i`** or **`j`**, but not both, so the test of the **`while`** loop will only be false for one of the two processes, not for both, and only one process will be able to enter the critical section, the other process will have to wait in the entry section.
        1. Which process enters the critical section first is then determined by the initial value of the **`turn`** variable.

Detailed Explanation on Example 1 (Continue)

2. Progress: No – If the **`turn`** variable contains the value **`j`** (for example) and P_i wants to enter the critical section and P_j does not want to enter the critical section (assuming there is no **`do{}while`** loop around P_j's code and P_j is busy doing something else) then P_i will wait for ever, even though P_j is not inside the critical section.
   1. P_i can only enter the critical section after P_j has been in the critical section, and P_j can only enter the critical section after P_i has been in the critical section.

      $P_i \rightarrow P_j \rightarrow P_i \rightarrow P_j \rightarrow P_i \rightarrow P_j \rightarrow P_i \rightarrow \cdots$
   2. If P_j does not want to enter the critical section again then P_i is going to get stuck and wait for ever in the entry section for P_j to exit the critical section, which P_j is never going to do.

      $P_i \rightarrow P_j \rightarrow P_i \rightarrow P_i$ again and gets stuck for ever.

Detailed Explanation on Example 1 (Continue)

3. Bounded Waiting: Yes - There is a bound of 1 time on the number of times that P_j (for example) is allowed to enter the critical section after P_i has made a request to enter its critical section and before that request is granted, because when P_j exits the critical section it changes the **`turn`** variable to be **`i`**, which then immediately allows P_i to enter the critical section.
   1. Even if we assume that P_j is very fast and P_i is very slow (for example), it is not possible for P_j to beat P_i to the race and exit the critical section and immediately re-enter it again before P_i is allowed to take its turn inside the critical section.
   2. In other words, it is not possible for P_j to prevent P_i from going into the critical section by going in and out of the critical section as fast as possible.
   3. So P_i is guaranteed to be able to enter the critical section at some point in the future (right after P_j is finished) and will not have to wait for ever, even if P_j wants to re-enter the critical section all the time.

Peterson's Solution

- It is a classic software-based solution to the critical-section problem
  - Good algorithmic description of solving the problem
- Solution for two processes by using two variables:
  - int turn; // indicates whose turn it is to enter the critical section.
  - Boolean flag[2] // indicate if a process is ready to enter the critical section.
    - flag[i] = *true* implies that process P_i is ready!
    - It is initialized to FALSE.

Algorithm for Process P_i & P_j

P_i

```
do {
      flag[i] = true; //ready
      turn = j; //allow Pj to enter
      while (flag[j] && turn = = j);
      critical section
      flag[i] = false; //exit
      remainder section
} while (true);
```

Ask for entry permission

Mutual Exclusion: Yes

Progress: Yes

Bounded waiting: Yes

P_j

```
do {
      flag[j] = true; //ready
      turn = i; //allow Pi to enter
      while (flag[i] && turn = = i);
      critical section
      flag[j] = false; //exit
      remainder section
} while (true);
```

Ask for entry permission

Synchronization Hardware

- Software-based solutions (such as Peterson's) are not guaranteed to work on modern computer architecture
- Many systems provide **hardware support** for implementing the critical section code.
  - **Uniprocessors** – could disable interrupts
    - Currently running code would execute without preemption
      - **too inefficient**
  - Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible,   the atomic hardware instruction will do the following work
      1. Test memory *word* and set value
      2. Swap contents of two memory *words*
    - E.g., test_and_set instruction in the next page

Atomic test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
        *target = TRUE;
        return rv:
}
```

1. The C code here is just to explain how the hardware instruction works.
2. This instruction is executed atomically by CPU as a single hardware instruction.
3. In practice, *target* is a pointer to the lock itself, shared by all the processes that want to acquire the lock.
   - If target if *FALSE*, the return value of *rv* is *FALSE*, means lock is *FALSE* (available), target's new value is *TRUE*
   - If *target* is true, the return value of *rv* is *TRUE*, means lock is *TRUE* (locked)

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock          entry section
    critical section
    release lock          exit section
    remainder section
} while (TRUE);
```

- Use the idea of locking
  - Protecting critical regions via locks
- A process that wants to enter the critical section must first get the lock.
- If the lock is already acquired by another process, the process will wait until the lock becomes available.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution using test_and_set:

```
do {
      while (test_and_set(&lock))
            ; /* do nothing */
      /* critical section */
      lock = FALSE;
      /* remainder section */
} while (true);
```

```
boolean test_and_set
 (boolean *target)
 {
    boolean rv = *target;
    *target = TRUE;
    return rv:
 }
```

```
do {
   acquire lock
   critical section
   release lock
   remainder section
} while (TRUE);
```

- When lock = true, keep while looping.
- When lock = FALSE, process can enter the critical section
- And set lock = TRUE, block other processes to enter.
- After finish the critical section, reset lock = FALSE, to allow other processes to enter the critical section.

Mutual Exclusion: Yes

Progress: Yes

Bounded waiting: No

Atomic compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

- The C code here is just to explain how the hardware instruction works.
- The instruction is executed atomically by CPU as a single hardware instruction.
  1. In practice, *value* is a pointer to the lock itself, shared by all the processes that want to acquire the lock.
  2. Set **value* (the lock) to the value of the passed parameter *new_value* but only if **value == expected*. That is, the swap takes place only under this condition.
  - Returns as result the original value of the lock.
- Similar to *test_and_set* but with an integer *lock* and an extra condition.

Solution using compare_and_swap

- Shared integer *lock* initialized to **0 (false)**;
- Solution using *compare_and_swap*:

```
do {
  while (compare_and_swap(&lock, 0, 1) != 0)
     ; /* do nothing */
   /* critical section */
   lock = 0;
   /* remainder section */

} while (true);
```

test_and_set(&lock)

Mutual Exclusion: Yes

Progress: Yes

Bounded waiting: No

Solution using compare_and_swap and test_and_set

- Mutual exclusion: Yes
- Progress: Yes
- Bounded Waiting: No
- Busy-waiting: Yes (in entry section, use while statement)
- Therefore the solution presented in the previous slide is not good enough either.

Bounded-waiting Mutual Exclusion with test_and_set

P_i

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

   /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;

   /* remainder section */
} while (true);
```

```
do {
      acquire lock
      critical section
      release lock
      remainder section
} while (TRUE);
```

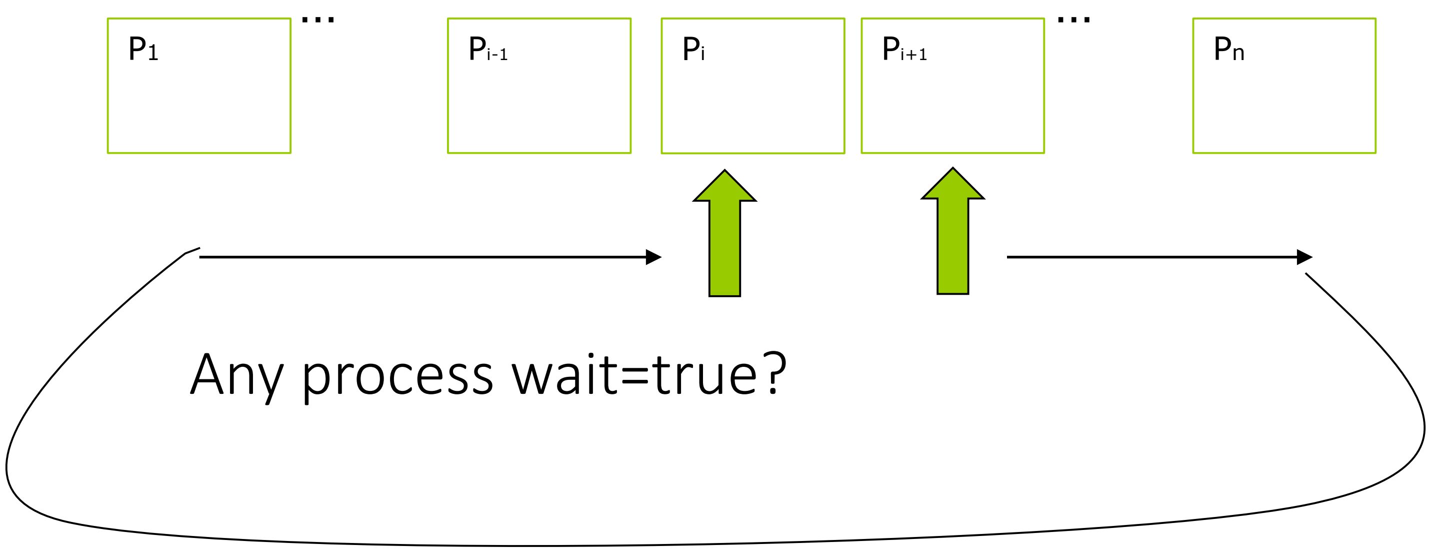
Common data structures:

boolean waiting[n];

boolean lock;

These data structures are initialized to **false**.

Bounded-waiting Mutual Exclusion with test_and_set (Continue)



Bounded-waiting Mutual Exclusion with test_and_set (Continue)

P_i

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

   /* critical section
*/

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;

   /* remainder section */
} while (true);
```

P_k

```
do {
   waiting[k] = true;
   key = true;
   while (waiting[k] && key)
      key = test_and_set(&lock);
   waiting[k] = false;

   /* critical section */

   j = (k + 1) % n;
   while ((j != k) && !waiting[j])
      j = (j + 1) % n;
   if (j == k)
      lock = false;
   else
      waiting[j] = false;

   /* remainder section */
} while (true);
```

Mutual exclusion: Yes   Progress: Yes   Bounded-waiting: Yes

Explanations on Three Reequipments for Slide 24

Mutual exclusion: Yes

```
waiting[i] = true;
key = true;
while (waiting[i] && key)
   key = test_and_set(&lock);
waiting[i] = false;
```

1. Process P_i can enter its critical section only if either waiting[*i*] == false or key == false
   - If waiting[*i*] is false, that means, another process finished critical section and give the turn to process P_i
   - If key is false, that means, lock was false before run test_and_set.
2. If another process (e.g., P_k) in waiting to enter, it cannot enter. Because waiting[*k*] is true, and lock is true.

Bounded-waiting Mutual Exclusion with test_and_set

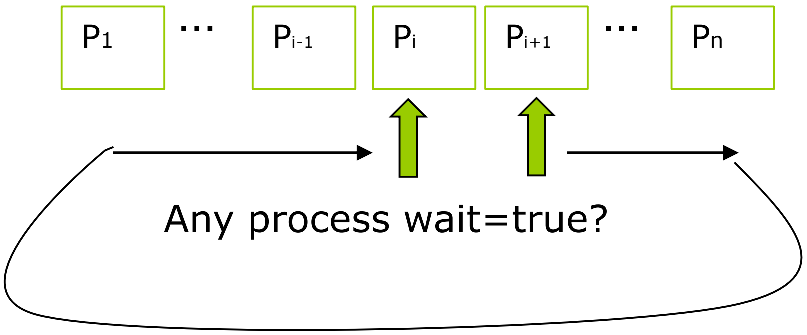
Progress: Yes

If P_i wants to enter the critical section, while other process do not want to enter, lock is false. So the test_and_set will return false.

Bounded waiting: Yes

If process P_i leaves the critical section, and another process (e.g. P_k) is waiting, the exit section in P_i will set waiting[*k*] to the false

```
j = (i + 1) % n;
while ((j != i) && !waiting[j])
  j = (j + 1) % n;
if (j == i)
    lock = false;
else
    waiting[j] = false;
```



Mutex Locks

- Previous hardware-based solutions are complicated and generally inaccessible to application programmers
- OS designers build high-level software tools to solve critical section problem
  - Simplest one of these tools is mutex lock(互斥锁)
    - Use mutex lock to protect a critical section by first `acquire()` a lock then `release()` the lock
    - Assumption: Calls to `acquire()` and `release()` must be atomic
      - Usually implemented via hardware atomic instructions
- This solution still requires busy waiting
  - This lock therefore is called a spinlock

acquire() and release()

```
acquire() {
     while (!available)
          ; /* busy wait */
     available = false;
}
```

```
release() {
          available = true;
}
```

```
do {
     acquire lock
          critical section
     release lock
          remainder section
} while (true);
```

- This solution requires busy waiting
  - This lock therefore called a spinlock(自旋锁)

POSIX Mutex Locks

```
#include <pthread.h>
pthread_mutex_t mutex;
pthread_mutex_init(&mutex,NULL);

pthread_mutex_lock(&mutex);

pthread_mutex_unlock(&mutex);
```

```
do {
    acquire lock
      critical section
    release lock
      remainder
section
 } while (true);
```

int pthread_mutex_init(pthread_mutex_t ***_mutex_, const pthread_mutexattr_t ***_mutexattr_**);**

The pthread_mutex_init() function initializes the specified mutex. If *attr* is non-NULL, the attributes specified are used to initialize the mutex. If the attribute object is modified later, the mutex's attributes are not affected. If *attr* is NULL, the mutex is initialized with default attributes, as specified for pthread_mutex_init().

A mutex can be statically initialized by assigning PTHREAD_MUTEX_INITIALIZER in its definition, as follows:
pthread_mutex_t def_mutex = PTHREAD_MUTEX_INITIALIZER;

A mutex must be initialized (either by calling pthread_mutex_init(), or statically) before it may be used in any other mutex functions.

Semaphore（信号灯）

- Semaphore is a synchronization tool
  - more sophisticated than mutex locks
- Semaphore *S:* an integer variable
  - S can only be accessed via two atomic operations
    - `wait()` and `signal()`
      - Originally called `P()` and `V()`

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

acquire lock

```
signal(S) {
    S++;
}
```

release lock

Semaphore（信号灯）

```
do {                                   Pi
    wait(S);
    /* critical section */
    signal(S)
    /* remainder section */
} while (true);
```

```
do {                                   Pj
    wait(S);
    /* critical section */
    signal(S)
    /* remainder section */
} while (true);
```

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

release lock

sem_wait and sem_post

```
#include <semaphore.h>
sem_t *sem_mutex;
```

Normally used among different processes

```
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

```
sem_wait(sem_mutex);  →  acquire lock
sem_post(sem_mutex);  →  release lock
```

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

POSIX Named Semaphore

#include <semaphore.h>
sem_t *sem_mutex;

<u>Normally used among different processes</u>

sem_t *sem_open(const char **name*, int *oflag*, mode_t *mode*, unsigned int *value*);

e.g., sem_mutex = sem_open("SEM", O_CREATE, 0666, 1);

multiple processes can easily use a common semaphore as a synchronization mechanism by the semaphore's name.

sem_wait(sem_mutex); → *acquire lock*

sem_post(sem_mutex); → *release lock*

```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

POSIX Unnamed Semaphore

```
#include <semaphore.h>
sem_t sem_mutex;
```

Normally used among different threads within a process

int sem_init(sem_t **sem*, int *pshared*, unsigned int *value*);

e.g., sem_init(&sem_mutex,0, 1);

Three parameters in sem_int:

1. A pointer to the semaphore
2. A flag indicating the level of sharing (shared between threads in a process or between processes)
3. The semaphore's initial value

sem_wait(&sem_mutex);

sem_post(&sem_mutex);

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

Semaphore Usage

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1
  - Same as a mutex lock
- Semaphore can be used to solve various synchronization problems

An example: Consider two processes P_1 and P_2 that require T_1 to happen before T_2. Solution:

```
semaphore synch;
synch = 0
```

```
P1:
    T1;
    signal(synch);
```

```
P2:
    wait(synch);
    T2;
```

Semaphore Implementation

- Advantage
  - implementation code is short
- Disadvantage :
  - Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
    - S--, S++ in wait() and signal()
    - Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
    - The solution now have busy waiting in critical section implementation
  - Both mutex lock and semaphore suffer from busy waiting.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
  - Each semaphore has:
    - an integer value
    - A queue of processes
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue

```
typedef struct{
      int value;
      struct process *list;
} semaphore;
```

Implementation with no Busy waiting (Cont.)

- Again, C code is just for explanation:

Busy waiting

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}


signal(S) {
    S++;
}
```

S: initial value 1

No busy waiting

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      //add this process to S->list;
      block();//put in a waiting queue
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
    //remove a process P from S->list;
      wakeup(P);//put in a ready queue
   }
}
```

S->value: initial value 1

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers-Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- Producer-Consumer problem
  - *n* buffers, each holds one item
  - Semaphore **mutex** initialized to the value 1
  - Semaphore **full** initialized to the value 0
  - Semaphore **empty** initialized to the value n

Bounded Buffer Problem (Cont.)

```
wait(S) {
  while (S <= 0)
    ; // busy
wait
  S--;
}

signal(S) {
    S++;
}
```

- The structure of the producer process

```
int n;
semaphore mutex = 1; //mutual exclusion for buffer pool access
semaphore empty = n; //number of empty entries in the buffer pool
semaphore full = 0; //number of items available in the buffer pool
do {
      ...
   /* produce an item in next_produced */
      ...
     wait(empty);
     wait(mutex);
          ...
     /* code to add next_produced to the
     buffer */
       ...
     signal(mutex);
     signal(full);
} while (true);
```

```
item next_produced;
while (true) {
     /* produce an item in
next_produced */
     while (counter ==BUFFER_SIZE)
        ;/* do nothing*/
     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
```

Bounded Buffer Problem (Cont.)

```
wait(S) {
  while (S <= 0)
    ; // busy wait
  S--;
}

signal(S) {
    S++;
}
```

- The structure of the consumer process

```
do{
  wait(full);
  wait(mutex);
       ...
    /* code to remove an item from buffer to
    next_consumed */
       ...
  signal(mutex);
  signal(empty);
       ...
 /* consume the item  in next_consumed */
       ...
} while (true);
```

```
item next_consumed;
while (true) {
  while (counter == 0)
    ; /* do nothing */
  next_consumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter--; //total # of item in the buffer
  /* consume the item in next_consumed */
}
```

Pthreads Synchronization

- Semaphore functions:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);

/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

Bounded Buffer Problem (Cont.)

Sample code demonstration in class

(Bounded Buffer using semaphores)

posix_bb.c

Bounded Buffer Problem (Cont.)

posix_bb.c

Compile command:
g++ -o posix_bb posix_bb.c -lpthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <deque>
#include <unistd.h>
#include <signal.h>

#define TRUE 1

/* functions for threads to call */
void *producer(void *param);  //for producer
void *consumer(void *param); //for consumer

//using three semaphores
sem_t  empty;
sem_t  full;
sem_t  mutex;

int shared_item = 0;

std::deque<int> myboundedbuffer;
pid_t pid;
```

Bounded Buffer Problem (Cont.)

posix_bb.c

```
int main(int argc, char *argv[]) {
      int i, scope;
      pthread_t producerID, consumerID; /* the thread identifier */
      pthread_attr_t attr;    /* set of attributes for the thread */
      int n = 10;
      pid = getpid();

      //create semaphores, and initialize
      sem_init(&empty, 0, n);//n
      sem_init(&full, 0, 0);//0
      sem_init(&mutex, 0, 1);//1

      /* get the default attributes */
      pthread_attr_init(&attr);

      /* create the threads */
      pthread_create(&producerID, &attr, producer, &producerID);
      pthread_create(&consumerID, &attr, consumer, &consumerID);

      /*Now join on each thread*/
      pthread_join(producerID, NULL);
      pthread_join(consumerID, NULL);
      return 0;
}
```

Bounded Buffer Problem (Cont.)

posix_bb.c

```
/*The thread will begin control in this function.*/
void *producer(void *param) {
        int id = *(int*)param;
        printf("producer Thread ID = %d\n", id);

        do {
                sem_wait(&empty);
                sem_wait(&mutex);

                shared_item++;
                if (shared_item > 20)
                        break;

                printf("Producer create an item %d\n", shared_item)
                myboundedbuffer.push_back(shared_item);

                sem_post(&mutex);
                sem_post(&full);
        }while (TRUE);

        kill(pid, SIGINT); //send a signal to kernel to terminate the process
        pthread_exit(0);
}
```

This program will finish when the producer has created 20 items

Bounded Buffer Problem (Cont.)

posix_bb.c

```
void *consumer(void *param) {

   int id = *(int*)param;
   printf("consumer Thread ID = %d\n", id);

  do {
          sem_wait(&full);
          sem_wait(&mutex);

          printf("\tConsumer process an item %d\n",  myboundedbuffer.front()) ;
          myboundedbuffer.pop_front();

          sem_post(&mutex);
          sem_post(&empty);

  } while (TRUE);

  pthread_exit(0);
}
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write data
- Problem
  - Allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared data
  - Data set
  - Semaphore **rw_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1
  - Integer **read_count** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process and reader processes

/* **rw_mutex**: a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section*/

```
semaphore rw_mutex = 1;
```

/* **mutex**: ensure mutual exclusion when the variable read_count is updated. */

```
semaphore mutex = 1;
```

/* **read_counter**: variable keeps track of how many processes are currently reading the object. */

```
int read_count = 0;
```

```
do {
     wait(rw_mutex);
               ...
    /* writing is performed */
               ...
    signal(rw_mutex);
} while (true);
```

For writer

```
do {
     wait(mutex);
     read_count++;
     if (read_count == 1)
          wait(rw_mutex);
     signal(mutex);
               ...
    /* reading is performed */
               ...
    wait(mutex);
    read_count--;
    if (read_count == 0)
         signal(rw_mutex);
    signal(mutex);
} while (true);
```

1st reader process

Last reader process

For reader

Readers-Writers Problem (Cont.)

Sample code demonstration in class
(Reader-Writer using semaphores)

posix_rw.c

Readers-Writers Problem (Cont.)

posix_rw.c

Compile command:

gcc -o posix_rw posix_rw.c -lpthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>
#include <signal.h>

#define TRUE 1

/* the thread runs in this function */
void *writer(void *param);
void *reader(void *param);

sem_t  rw_sem;
sem_t  mutex;
int read_count = 0;

int shared_data = 0;
pid_t pid;
```

Readers-Writers Problem (Cont.)

posix_rw.c

```
int main(int argc, char *argv[]) {
        int i, scope;
        pthread_t writerID, readerID1, readerID2;        /* the thread identifier */
        pthread_attr_t attr;              /* set of attributes for the thread */

        pid = getpid();

        //create semaphores, and initialize to 1
        sem_init(&rw_sem, 0, 1);
        sem_init(&mutex, 0, 1);

        /* get the default attributes */
        pthread_attr_init(&attr);

        /* create 3 threads */
        pthread_create(&writerID, &attr, writer, &writerID);
        pthread_create(&readerID1, &attr, reader, &readerID1);
        pthread_create(&readerID2, &attr, reader, &readerID2);

        /* Now join on each thread */
        pthread_join(writerID, NULL);
        pthread_join(readerID1, NULL);
        pthread_join(readerID2, NULL);
        return 0;
}
```

Readers-Writers Problem (Cont.)

posix_rw.c

```
void *writer(void *param) {
        int id = *(int*)param;
        printf("Writer Thread ID = %d\n", id);
        shared_data = 0;
        do {
                sem_wait(&rw_sem);
                shared_data++;
                printf("Writer:  new data = %d\n", shared_data);
                sem_post(&rw_sem);
                sleep(1);
        } while (TRUE);

        pthread_exit(0);
}
```

Readers-Writers Problem (Cont.)

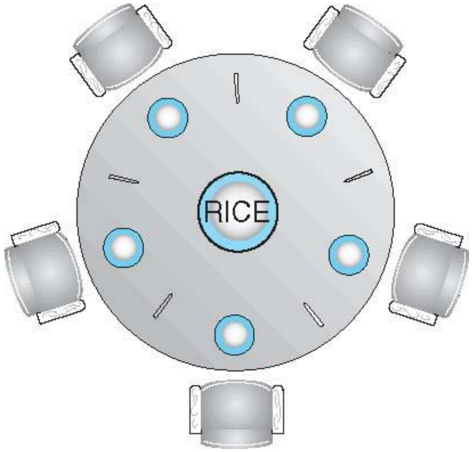
posix_rw.c

```
void *reader(void *param) {
        int id = *(int*)param;
        do {
                sem_wait(&mutex);
                read_count++;
                if (read_count == 1)
                        sem_wait(&rw_sem);
                sem_post(&mutex);

                printf("\tReader_TID(%d): data(%d)\n", id, shared_data);

                if (shared_data > 10)
                        break;

                sem_wait(&mutex);
                read_count--;
                if (read_count == 0)
                        sem_post(&rw_sem);
                sem_post(&mutex);
                sleep(1);
        } while (TRUE);

        kill(pid, SIGINT); //send a signal to kernel to terminate the process
        pthread_exit(0);
}
```

Readers-Writers Problem Variations

- *First* variation – no reader kept waiting unless writer has permission to use shared object
- *Second* variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing **reader-writer locks**

Dining-Philosophers Problem

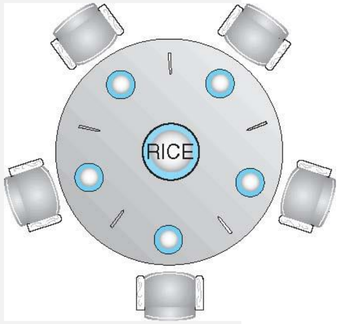


- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both chopsticks to eat, then release both when done
- In the case of 5 philosophers
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher ***i***:

```
do {
        wait (chopstick[i] );                   //left chopstick
        wait (chopstick[ (i + 1) % 5] );   //right chopstick
        //  eat state
        signal (chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
        //  think state
} while (TRUE);
```



- What is the problem with this algorithm?
  - Although this solution guarantees that no two neighbors are eating simultaneously, but it could create a deadlock.
    - Suppose that all five philosophers become hungry at the same time and each grabs the left chopstick. No chopstick left. When each philosopher tries to grab the right chopstick, he/she will be delayed forever.
    - Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock solutions
  1. Allow at most 4 philosophers to be sitting simultaneously at the table.
  2. Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section.)
  3. Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.
- A deadlock-free solution using monitor is provided later (appendix slide 74-80).
- Any satisfactory solution must guard against the possibility that one of the philosophers may starve to death.
- A deadlock-free does not necessarily eliminate the possibility of starvation.

Problems with Semaphores

- Deadlock and starvation are possible.
- Incorrect use of semaphore operations:
  - signal (mutex) .... wait (mutex)
  - wait (mutex) ... wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- It is the responsibility of the application programmer to use semaphores correctly.

Deadlock and Starvation

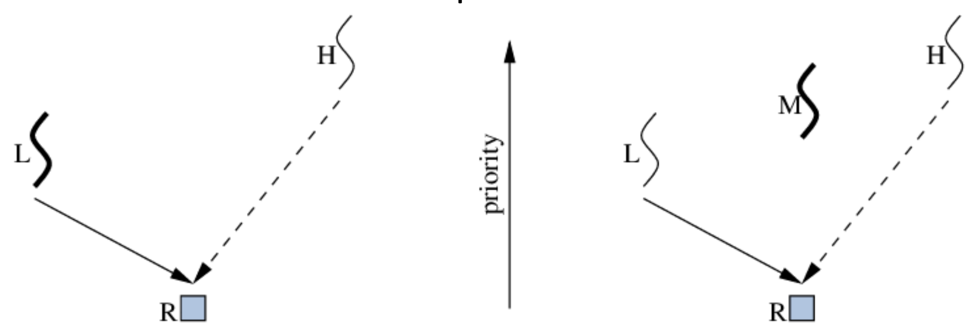
- Let S and Q be two semaphores initialized to 1

```
      P0                        P1
   wait(S);                  wait(Q);
   wait(Q);                  wait(S);
   ...                       ...
   signal(Q);                signal(S);
   signal(S);                signal(Q);
```

- Can cause starvation and priority inversion
  - Starvation – indefinite blocking
    - A process may never be removed from the semaphore queue in which it is suspended)
  - Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Deadlock and Starvation

- **priority inversion example:**
  - Assume three processes
    - *L*, *M*, and *H with* the priority order: *L* < *M* < *H*.
  - Assume that
    - process *H* requires resource *R*, which is currently hold by process *L*.
    - process *H* would wait for *L* to finish using resource *R*.
  - However, process *M* becomes runnable, and preempts process *L*.
  - Consequence:
    - process *M* (with middle priority) has affected how long process *H* must wait for *L* to relinquish resource *R*.



priority inversion

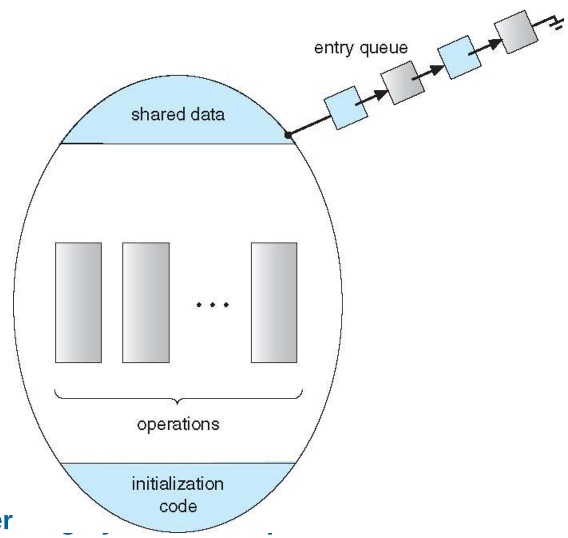
unbounded priority inversion

Deadlock and Starvation

- Solution
  - Use priority-inheritance protocol
    - All processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.
    - When they are finished, their priorities revert to their original values.
  - This protocol solve the previous priority inversion problem

Monitors

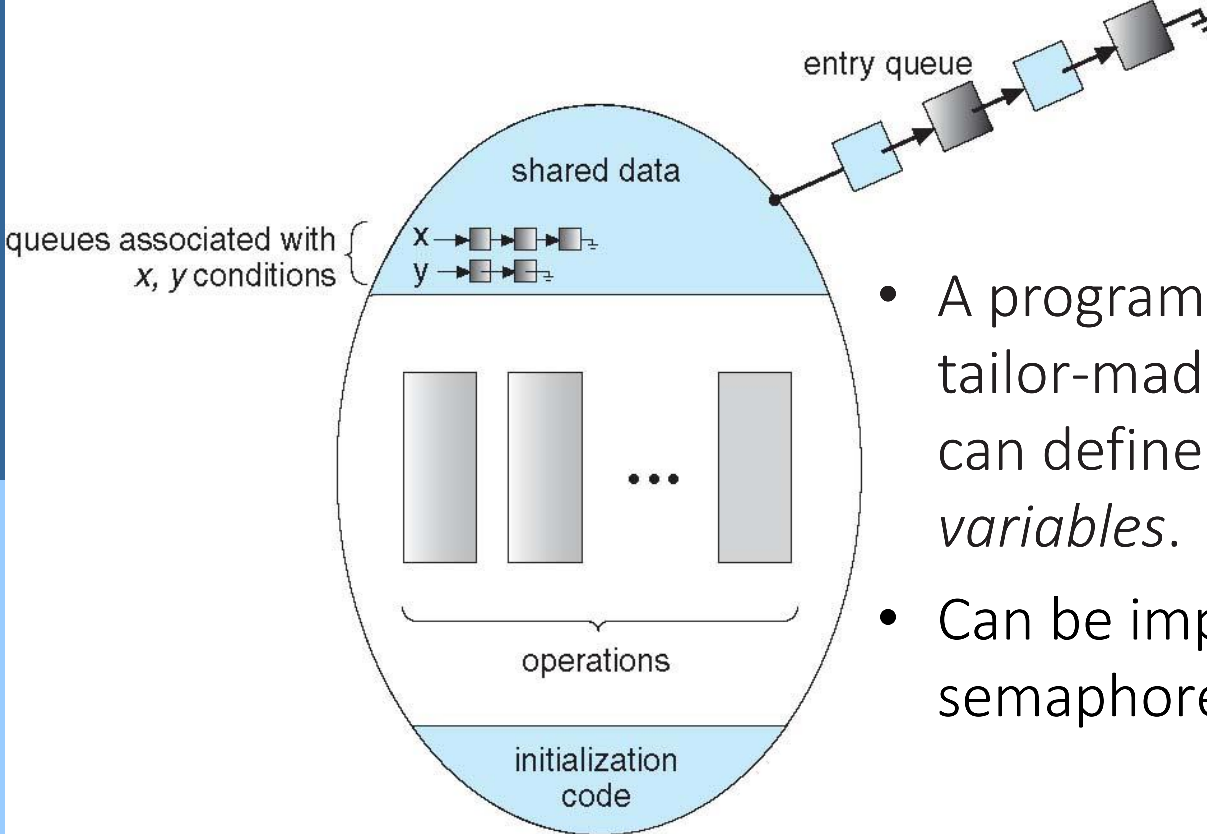
- Monitor (管程)
  - A high-level abstraction that provides a convenient and effective mechanism for process synchronization
    - *Abstract data type*, internal variables only accessible via procedures
    - Only one process may be active within the monitor at a time
    - Can utilize condition variables to suspend or resume processes




```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (…) { …. }
        procedure Pn (…) {……}
        Initialization code (…) { … }
}
```

Monitors

https://baike.baidu.com/item/%E7%AE%A1%E7%A8%8B/10503922?fr=aladdin



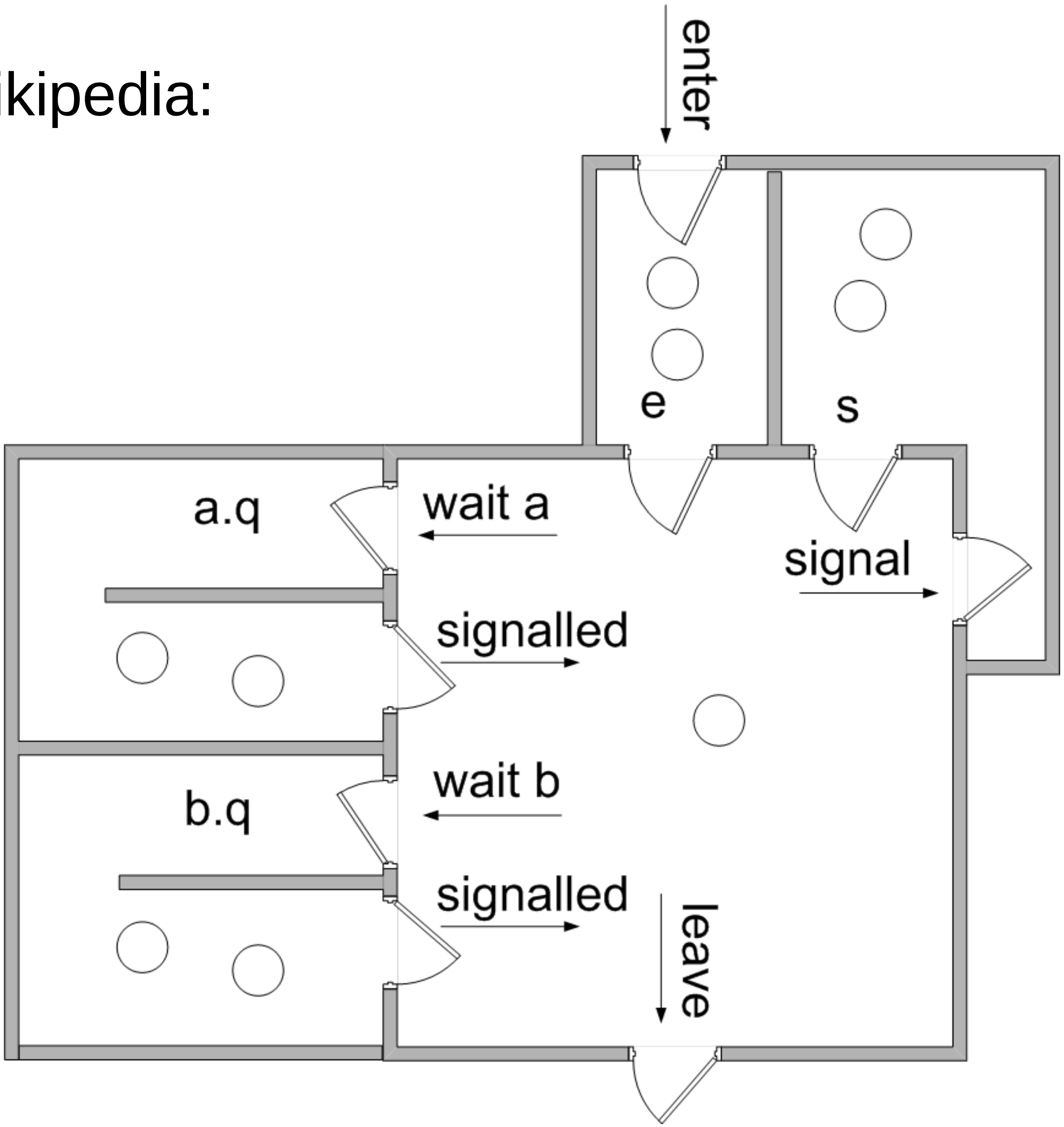
- A programmer who needs to write a tailor-made synchronization scheme can define one or more condition *variables*.
- Can be implemented with semaphores

- condition x, y;
- Two operations are allowed on a condition variable:
  - x.wait() – a process that invokes the operation is suspended until x.signal()
  - x.signal() – resumes one suspended process (if any)

    If no x.wait() on the variable, then it has no effect on the variable

Monitor with Condition Variables

From Wikipedia:



End of Chapter 6 & 7

Appendices

- The appendix parts are for students who are interested in knowing more about the programming related to Dining Philosophers using Dining Philosophers using Monitor and Monitor implemented using semaphores introduced in this lecture.

Monitors

https://en.wikipedia.org/wiki/Monitor_(synchronization)

In concurrent programming, a **monitor** is a synchronization construct that allows threads to have both **mutual exclusion** and the ability to **wait** (block) for a certain condition to become true.

Monitors also have a mechanism for signaling other threads that their condition has been met.

A monitor consists of a **mutex (lock)** object and **condition variables**.

A **condition variable** is basically a **container of threads** that are waiting for a certain condition.

Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.
Another definition of **monitor** is a **thread-safe** class, object, or module that uses wrapped mutual exclusion in order to safely allow access to a method or variable by more than one thread.

The defining characteristic of a monitor is that its methods are executed with **mutual exclusion**: At each point in time, at most one thread may be executing any of its methods.

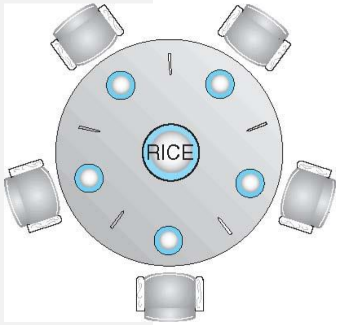
By using one or more condition variables it can also provide the ability for threads to wait on a certain condition.

Condition Variables Choices

- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in parallel. (only one process can access) If Q is resumed, then P must wait.
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons (**优缺点**) – language implementer can decide
  - Monitors implemented in **Concurrent Pascal** compromise
    - P executing signal immediately leaves the monitor, and Q is immediately resumed
      - i.e., signal() is the last sentence
  - Monitor is Implemented in other languages including C#, Java

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers {
    enum { THINKING; HUNGRY, EATING) state[5] ;
    condition self[5];
    void pickup (int i) {
           state[i] = HUNGRY;
           test(i);
           while (state[i] != EATING) self[i].wait();
    }
    void putdown (int i) {
            state[i] = THINKING;
            test((i+4)%5);//left
            test((i+1)%5);//right
    }
    void test (int i) {
            if ((state[(i+4)%5] != EATING) && (state[i] == HUNGRY)
    &&
            (state[(i+1)%5] != EATING) ) {
                 state[i] = EATING ;
                 self[i].signal();
            }
    }
     initialization_code() {
           for (int i = 0; i < 5; i++)
                  state[i] = THINKING;
    }
}
```



Solution to Dining Philosophers (Cont.)

- Each philosopher i invokes the operations pickup() and putdown() in the following sequence:

```
while(true) {

        DiningPhilosophers.pickup(i);

        EAT();

        DiningPhilosophers.putdown(i);
        Think();

}
```

- No deadlock, but starvation is possible

Code demonstration in class

Dining Philoshophers using Monitor

posix_dpm1.c

Solution to Dining Philosophers (Cont.)

Filename: posix_dpm1.c
compile command:g++ -o dpm1 posix_dpm1.c -lpthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>

const int N = 5;
int phil_num[N]={0,1,2,3,4};
int eat_count[N] = { 0, 0, 0, 0, 0 };
enum State { THINKING, HUNGRY, EATING};

void *philosopher(void *num);
```

Solution to Dining Philosophers (Cont.)

```
//monitor struct for dining philosophers implemented with pthread condition variable
typedef struct Monitor {
public:
        int state[N];
        pthread_cond_t self[N];      //pthread condition variables
        pthread_mutex_t  mutex_lock;
        void pickup(int i) {         //external function
                pthread_mutex_lock(&mutex_lock);
                state[i] = HUNGRY;
                test(i);
                while (state[i] != EATING) {
                        pthread_cond_wait(&self[i], &mutex_lock);
                }
                pthread_mutex_unlock(&mutex_lock);
        }

        void putdown(int i) {        //external function
                pthread_mutex_lock(&mutex_lock);
                state[i] = THINKING;
                test((i + N-1) % N);         //left
                test((i + 1) % N);           //right
                pthread_mutex_unlock(&mutex_lock);
        }

        void test(int i) {//internal function
                if ((state[(i + N-1) % N] != EATING) && (state[i] == HUNGRY) &&
                        (state[(i + 1) % N] != EATING)) {
                        state[i] = EATING;
                        pthread_cond_signal(&self[i]);
                }
        }

        void initialization()  {
                pthread_mutex_init(&mutex_lock, NULL);
                printf("use pthread conditional variables\n");
                for (int i = 0; i < N; i++){
                        state[i] = THINKING;
                        pthread_cond_init(&self[i], NULL);
                }
        }
} DiningPhilosophers;
```

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
```

Solution to Dining Philosophers (Cont.)

```
DiningPhilosophers dp;

int main() {
        int i;
        pthread_t thread_id[N];

        dp.initialization();

        for(i = 0; i < N; i++)
                pthread_create(&thread_id[i],NULL,philosopher,&phil_num[i]);

        for(i = 0; i < N; i++)
                pthread_join(thread_id[i],NULL);

        for (i = 0; i < N; i++)
                printf("Philosopher %d eat %d times\n", i + 1, eat_count[i]);

        return 0;
}
```

Solution to Dining Philosophers (Cont.)

```
void *philosopher(void *num) {
        int loops = 0;
        int i = *(int*)num;
        while (loops < 5) {

                //hungry and want to eat
                printf("Philosopher %d is hungry\n",i+1);
                dp.pickup(i);

                //eating time
                printf("\tPhilosopher %d is eating\n",i+1);
                eat_count[i]++;
                sleep(2);

                 //putting down forks
                dp.putdown(i);
                printf("\t\tPhilosopher %d is putting down forks\n",i+1);

                //thinking time
                printf("Philosopher %d is thinking\n",i+1);
                sleep(1);
                ++loops;
        }
        pthread_exit(0);
```

Monitor Implementation Using Semaphores

- Variables
  - semaphore mutex; // (initially = 1)
    - A process must
      - execute wait(mutex) before entering the monitor and
      - execute signal(mutex) after leaving the monitor.
  - semaphore next;   // (initially = 0)
    - A signaling process must wait until the resumed process either leaves or waits, an additional semaphore, `next` is used.
    - The signaling processes can use `next` to suspend themselves.
  - int next_count = 0;
    - count the number of processes suspended on semaphore `next`.

Monitor Implementation Using Semaphores

- Each external procedure *F* will be replaced by

```
wait(mutex);
……
 body of F;

if (next_count > 0)//some processes suspended
  signal(next)
else
  signal(mutex);
```

Mutual exclusion within a monitor is ensured

Monitor Implementation – Condition Variables

- For each condition variable *x*, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- Th

```
x_count++;                 // Because P is adding itself to the wait queue of x.
if (next_count > 0)        // If other process in global wait queue
    signal(next);          // allow it to execute inside monitor
else
    signal(mutex);         // allow other process to enter monitor;
wait(x_sem);               // then P falls asleep in wait queue for x.
                           // Some time later another process calls x.signal
x_count--;                 // then P leaves the wait queue for x after waking up.
```

Monitor Implementation (Cont.)

- The operation **`x.signal`** can be implemented as:

```
if (x_count > 0) { // If there is a Q in the wait queue for x:
    next_count++;   // P is going to add itself to the global wait queue
    signal(x_sem);  // P wakes up Q
    wait(next);     // P falls asleep in the global wait queue
                    // Some time later another process calls signal(next)
    next_count--;   // then P leaves the global wait queue after waking up.
}
```

This implementation is applicable to the definitions of monitors given by both Hoare and Brinch-Hansen.

Resuming Processes within a Monitor

- If several processes queued on condition `x`, and `x.signal()` executed, which should be resumed?
  - FCFS (First Come First Served; FIFO) frequently not adequate
  - Priority based: conditional-wait construct of the form `x.wait(c)`
    - Where `c` is priority number
    - When `x.signal()` is called, the process in the wait queue of `x` which has the lowest number (highest priority) is going to be woken up first.

To illustrate this new mechanism, consider the ResourceAllocator monitor shown in next slide, which controls the allocation of a single resource among competing processes.

Code demonstration in class

Dining Philosophers using Monitor
implemented using semaphores

posix_dpm2.c

Code demonstration in class

```
/**
 * A pthread program illustrating POSIX dining-philosophers problem
 * using monitor struct and  implementation using semaphores
 *
 * To compile:
 *        g++ -o dpm2 posix_dpm2.c  -lpthread
 */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>

const int N = 5;
int phil_num[N]={0,1,2,3,4};
int eat_count[N] = {0,0,0,0,0};
enum Philosophers_State { THINKING, HUNGRY, EATING};

void *philosopher(void *num);
```

Code demonstration in class (Cont.)

```
//monitor struct for dining philosophers
typedef struct Monitor {
        int state[N];
        //use semaphore to implement condition variables
        sem_t x_sem[N];
        int x_count[N];
        sem_t mutex;
        sem_t next;
        int next_count;//count the number of threads suspended on next

        //wait function for condition variable x_sem
        void wait(int i) {
                x_count[i]++;
                if (next_count > 0)
                        sem_post(&next);
                else
                        sem_post(&mutex);
                sem_wait(&x_sem[i]);
                x_count[i]--;
        }

        //signal function for condition variable x_sem
        void signal(int i){
                if (x_count[i] > 0){
                        next_count++;
                        sem_post(&x_sem[i]);
                        sem_wait(&next);
                        next_count--;
                }
        }

        //external function
        void pickup(int i) {
                sem_wait(&mutex);
                state[i] = HUNGRY;

                test(i);

                while (state[i] != EATING)
                        wait(i);

                if (next_count > 0)
                        sem_post(&next);
                else
                        sem_post(&mutex);
}

        //external function
        void putdown(int i) {
           sem_wait(&mutex);

           state[i] = THINKING;

           test((i + N-1) % N);//left
           test((i + 1) % N);//right

           if (next_count > 0)
                   sem_post(&next);
           else
                   sem_post(&mutex);
        }

        //internal function
        void test(int i) {
           if ((state[(i + N-1) % N] != EATING) && (state[i] == HUNGRY)
              && (state[(i + 1) % N] != EATING)) {
                   state[i] = EATING;
                   signal(i);
           }
        }

        void initialization()  {
           printf("use semaphore to implement condition variables\n");
           for (int i = 0; i < N; i++) {
                   state[i] = THINKING;
                   x_count[i] = 0;
                   sem_init(&x_sem[i], 0, 1);//1
           }
           next_count = 0;
           sem_init(&next, 0, 0);//0
           sem_init(&mutex, 0, 1);
        }
} DiningPhilosophers;
```

Code demonstration in class (Cont.)

```
DiningPhilosophers dp;

int main() {
        int i;
        pthread_t thread_id[N];

        dp.initialization();

        for(i = 0; i < N; i++)
            pthread_create(&thread_id[i],NULL,philosopher,&phil_num[i]);

        for(i = 0; i < N; i++)
            pthread_join(thread_id[i],NULL);

        for (i = 0; i < N; i++)
            printf("Philosopher %d eat %d times\n", i + 1, eat_count[i]);
        return 0;
}
```

Code demonstration in class (Cont.)

```
void *philosopher(void *num) {
        int i = *(int*)num;
        int loops = 0;
        while (loops < 5) {

                //hungry and want to eat
                printf("Philosopher %d is hungry\n",i+1);
                dp.pickup(i);

                //eating time
                printf("\tPhilosopher %d is eating\n",i+1);
                eat_count[i]++;
                sleep(2);

                 //putting down forks
                dp.putdown(i);
                printf("\t\tPhilosopher %d is putting down forks\n",i+1);

                //thinking time
                printf("Philosopher %d is thinking\n",i+1);
                sleep(1);
                ++loops;
        }
        pthread_exit(0);
}
```

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator {
   boolean busy;
   condition x;
   void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
   }
   void release() {
        busy = FALSE;
        x.signal();
   }
   initialization code() {
        busy = FALSE;
   }
}
```

```
monitor ResourceAllocator {
   boolean busy;
   pthread_cond_t x;
   void acquire(int time) {
      if (busy)
          pthread_cond_wait(&x, NULL);
      busy = TRUE;
   }
   void release() {
       busy = FALSE;
       pthread_cond_signal(&x);
   }
   initialization code() {
       pthread_cond_init(&x, NULL);
       busy = FALSE;
   }
}
```

Synchronization Examples

- Windows
- Linux
- Pthreads

Synchronization Examples

- Windows
- Inside kernel
  - Uses interrupt masks () to protect access to global resources on uniprocessor systems
  - Uses spinlocks on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Outside kernel
  - Provides dispatcher objects(event, mutex, semaphore, timer, and thread) for thread synchronization
    - Synchronization using mutexes, semaphores, events, and timers
      - Events
        - An event acts much like a condition variable
      - Timers notify one or more thread when time expired
  - A document online: Introduction to Kernel Dispatcher Objects - Windows drivers | Microsoft Docs

Windows Synchronization

- (Mutex)Dispatcher objects either signaled-state (object available, thread will not block) or non-signaled state (thread will block)

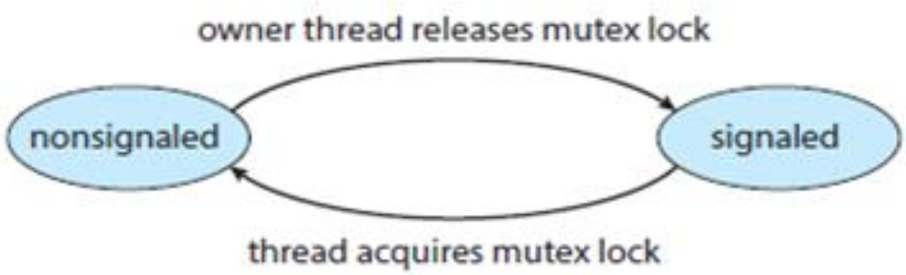


Figure 7.8 Mutex dispatcher object.

If a thread tries to acquire a mutex dispatcher object that is in a non-signaled state, that thread will be suspended and placed in awaiting queue for the mutex object. When the mutex moves to the signaled state (because another thread has released the lock on the mutex), the thread waiting at the front of the queue will be moved from the waiting state to the ready state and will acquire the mutex lock.

Windows Synchronization Cont.

- A critical-section object is a user-mode mutex that can be used without kernel intervention.
  - Document: [win32/critical-section-objects.md at docs · MicrosoftDocs/win32 · GitHub]

A *critical section object* provides synchronization similar to that provided by a mutex object, except that a critical section can be used only by the threads of a single process. Critical section objects cannot be shared across processes.

Event, mutex, and semaphore objects can also be used in a single-process application, but critical section objects provide **a slightly faster, more efficient mechanism for mutual-exclusion synchronization** (a processor-specific test and set instruction). Like a mutex object, a critical section object can be owned by only one thread at a time, which makes it useful for protecting a shared resource from simultaneous access. Unlike a mutex object, there is no way to tell whether a critical section has been abandoned.

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive(抢占式)
- Linux provides:
  - atomic integers
    - all math operations using atomic integers are performed without interruption
  - Mutex locks
  - Spinlocks and semaphores
  - reader-writer versions of Spinlocks and semaphores

Linux Synchronization Cont.

- On single-cpu system, spinlocks are replaced by enabling and disabling kernel preemption

Single Processor	Multiple Processors
Disable kernel preemption	Acquire spin lock
Enable kernel preemption	Release spin lock

```
Linux provides two simple system calls:
        preempt_disable()
        preempt_enable()
for disabling and enabling kernel
preemption
```

when a lock is held for a short duration, **spinlocks & enabling and disabling kernel preemption** are used;

When a lock must be held for a longer period, **semaphores or mutex locks** are used.

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable

```
#include <pthread.h>
pthread_mutex_t mutex;
pthread_cond_t cond_var;

//create a mutex lock
pthread_mutex_init(&mutex,NULL);
//create a condition
pthread_cond_init(&cond var,NULL);

pthread_cond_wait(&cond, NULL);
pthread_cond_signal(&cond);

/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

Pthreads Synchronization

- Non-portable extensions include:
  - read-write locks
  - Spinlocks
  - semaphore

Pthreads Synchronization

- Read-write locks functions

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *rwptr, const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwptr);

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(pthread_rwlockatttr_t *attr);

int pthread_rwlock_rdlock(pthread_rwlock_t *rwptr);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwptr);
int pthread_rwlock_unlock(pthread_rwlock_t *rwptr);
```

- Spinlocks functions

```
#include <pthread.h>

int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

Pthreads Synchronization

- Semaphore functions:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);

/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

Alternative Approaches

- Explore various features provided in both programming languages and hardware that support designing thread-safe concurrent applications.
  - Transactional Memory
  - OpenMP
  - Functional Programming Languages

Transactional Memory

- A memory transaction is a sequence of read-write operations to memory that are performed atomically.

```
void update(){
        acquire();
        /* read/write memory */
        release();
}
```

Deadlock is possible

```
void update(){
        atomic{
                /* read/write memory */
        }
}
```

(1) The transactional memory system is responsible for guaranteeing atomicity.

(2) Because no locks are involved, deadlock is not possible.

OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```
        void update(int value)
    {
            #pragma omp critical
            {
                    count += value
            }
    }
```

- The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.

Functional Programming Languages

- Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.
- Variables are treated as immutable (不变的) and cannot change state once they have been assigned a value.
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.

Summary

- Given a collection of cooperating sequential processes that share data, mutual exclusion must be provided to ensure that a critical section of code is used by only one process or thread at a time.
- Hardware provides several operations that ensure mutual exclusion, but too complicated for most developers to use.
- Mutex locks and semaphores: to solve various synchronization problems and can be implemented efficiently, especially atomic operations are available.
- Various synchronization problems are discussed:
  1. bounded-buffer problem
  2. readers–writers problem
  3. dining-philosophers problem

Summary cont.

- **Monitors** provide a synchronization mechanism for **sharing abstract data types**.
- A **condition variable** provides a method by which **a monitor function can block its execution until it is signaled to continue**.
- OS provides support for synchronization. For example,Windows, Linux, provide mechanisms such as **semaphores**, **mutex** locks, **spinlocks**, and **condition variables** to control access to shared data.
- The Pthreads API provides support for **mutex locks** and **semaphores**, as well as **condition variables**.
- Several alternative approaches focus on synchronization for multicore systems.
  - using transactional memory;
  - using the compiler extensions offered by OpenMP;