Lecture 7

Process Synchronization II

Lecture Information

Ceng328 *Operating Systems* at April 06, 2010

Dr. Cem Özdoğan
Computer Engineering Department
Çankaya University

Contents

Peterson's Solution I

- A classic software-based solution to the critical-section problem known as Peterson's solution.
- Does not require strict alternation.
- Peterson's solution requires two data items to be shared between the two processes:

```
int turn;
boolean flag[2];
```

  - The variable **turn** indicates whose turn it is to enter its CS. That is, if *turn* $== i$, then process P_i is allowed to execute in its CS.
  - The **flag array** is used to indicate if a process is ready to enter its CS. For example, if *flag*[i] is true, this value indicates that P_i is ready to enter its CS.

Peterson's Solution II

- The algorithm for Peterson's solution is seen in Fig. 1.

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Figure: The structure of process P_i in Peterson's solution.

- **Mutual exclusion is preserved.**
- **The progress requirement is satisfied & The bounded-waiting requirement is met.**
- Burns CPU cycles; requires busy waiting
- It can be extended to work for *n* processes, but overhead.

Peterson’s Solution III

```
#define FALSE  0
#define TRUE   1
#define N      2                          /* number of processes */

int turn;                                 /* whose turn is it? */
int interested[N];                        /* all values initially 0 (FALSE) */

void enter_region(int process);           /* process is 0 or 1 */
{
     int other;                           /* number of the other process */

     other = 1 − process;                 /* the opposite of process */
     interested[process] = TRUE;          /* show that you are interested */
     turn = process;                      /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)            /* process: who is leaving */
{
     interested[process] = FALSE;         /* indicate departure from critical region */
}
```

Figure: Peterson’s solution for achieving mutual exclusion.

Peterson's Solution IV

- **Sleep and wakeup.** Peterson's solution has not only the defect of requiring **busy waiting** but it can also have unexpected effects;
  - Consider a computer with two processes, *H*, with high priority and *L*, with low priority.
  - The scheduling rules are such that *H* is run whenever it is in ready state.
  - At a certain moment, with *L* in its critical region, *H* becomes ready to run (e.g., an I/O operation completes).
  - *H* now begins busy waiting, but since *L* is never scheduled while H is running, *L* never gets the chance to leave its critical region, so *H* loops forever.
  - This situation is sometimes referred to as the *priority inversion problem*.
- IPC primitive that **blocks** instead of wasting CPU time (while loop) when they are not allowed to enter their CRs.
- One of the simplest is the pair **sleep** and **wakeup**.
  - *Sleep* is a system call that causes the caller to block, that is, be suspended until another process wakes it up.
  - The *wakeup* call has one parameter, the process to be awakened.

Semaphores I

- A synchronization tool called **semaphore**.
- Semaphores are variables that are used to *signal* the status of shared resources to processes.
- Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use.
- A semaphore *S* is accessed only through two standard atomic operations (apart from initialization): *wait*() (sleep) and *signal*() (wakeup).
- A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.
- Two operations, **down** and **up** (generalizations of sleep and wakeup, respectively);

The definition of *wait*() is as follows:

```
wait (S) {
  while S <= 0
    ;// no-op
  S--;
}
```

The definition of *signal*() is as follows:

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


Semaphores II

- All the modifications to the integer value of the semaphore in the *wait*() and *signal*() operations must be executed **indivisibly** (atomicity).
- That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In addition, in the case of *wait*(S),
  - the testing of the integer value of S ($S \leq 0$),
  - and its possible modification ($S--$),

  must also be executed without interruption.

Usage I

- **Counting and binary semaphores.** The value of a counting semaphore can range over an unrestricted domain.
- The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as **mutex locks**, as they are locks that provide mutual exclusion.
- Counting semaphores can be used *to control access to a given resource* consisting of a finite number of instances.
  - The semaphore is initialized to the number of resources available.
  - Each process that wishes to use a resource performs a *wait*() operation on the semaphore (thereby decrementing the *count*).
  - When a process releases a resource, it performs a *signal*() operation (incrementing the *count*).
  - When the *count* for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Usage II

- We can use binary semaphores to deal with the critical-section problem for multiple processes.
- The *n* processes share a semaphore, **mutex**, initialized to 1.
- Each process P_i is organized as shown in Fig. 3.

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

Figure: Mutual-exclusion implementation with semaphores.

Implementation I

- The main disadvantage of the semaphore is that it requires busy waiting.
  - While a process is in its CS, any other process that tries to enter its CS must loop continuously in the entry code.
  - Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called a **spinlock** because the process “spins” while waiting for the lock (context switch is not required).
- To overcome the need for busy waiting, we can modify the definition of the *wait*() and *signal*() semaphore operations.
  - When a process executes the *wait*() operation and finds that the semaphore value is not positive, it must wait.
  - However, rather than engaging in busy waiting, the process can **block** itself.
  - The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
  - Then control is transferred to the CPU scheduler, which selects another process to execute.

Implementation II

- A process that is blocked, waiting (sleeping) on a semaphore *S*, should be restarted when some other process executes a *signal*() (wakeup) operation.
- It changes the process from the waiting state to the ready state.
- The list of waiting processes can be easily implemented by a link field in each process control block (PCB).
- The critical aspect of semaphores is that they be executed **atomically**.
- We must guarantee that no two processes can execute *wait*() and *signal*() operations on the same semaphore at the same time.

Deadlocks and Starvation I

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
- The event in question is the *execution of a signal*() *operation*.
- When such a state is reached, these processes are said to be **deadlocked**.
- Another problem related to deadlocks is **indefinite blocking**, or **starvation**, a situation in which processes wait indefinitely within the semaphore.
- Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Deadlocks and Starvation II

- To illustrate deadlock, we consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, S and Q, set to the value 1:

```
   P0              P1

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

  - Suppose that P_0 executes *wait*(S) and then P_1 executes *wait*(Q).
  - When P_0 executes *wait*(Q), it must wait until P_1 executes *signal*(0).
  - Similarly, when P_1 executes *wait*(S), it must wait until P_0 executes *signal*(S).
  - Since these *signal*() operations cannot be executed, P_0 and P_1 are deadlocked.

Mutexes I

- A **mutex** is a variable that can be in one of two states: *unlocked* or *locked*.
- Two procedures are used with mutexes.
  - When a thread (or process) needs access to a critical region, it calls *mutex_lock*.
  - If the mutex is currently unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.
  - On the other hand, if the mutex is already locked, the calling thread is <u>blocked</u> until the thread in the critical region is finished and calls *mutex_unlock*.
- If multiple threads are blocked on the mutex, one of them is *chosen at random* and allowed to acquire the lock.
- With threads, there is no clock that stops threads that have run too long.

Mutexes II

- Consequently, a thread that tries to acquire a lock by busy waiting will loop forever and never acquire the lock because it never allows any other thread to run and release the lock.
- That is where the difference between *enter_region* and *mutex_lock* comes in.
- When the later fails to acquire a lock, it calls *thread_yield* to give up the CPU to another thread.
- Consequently there is no busy waiting. When the thread runs the next time, it tests the lock again.

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Figure: Some of the Pthreads calls relating to the mutexes.

Mutexes III

```c
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                     /* put item in buffer */
                pthread_cond_signal(&condc);    /* wake up consumer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}

void *consumer(void *ptr)                       /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                     /* take item out of buffer */
                pthread_cond_signal(&condp);    /* wake up producer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}
```

Figure: Using threads to solve the producer-consumer problem I.

Mutexes IV

```
int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Figure: Using threads to solve the producer-consumer problem II.

The Bounded-Buffer Problem I

- We assume that the pool consists of *n* buffers.
- The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.
- The *empty* (initially *n*) and *full* (initially 0) semaphores count the number of empty and full buffers.
- The code for the producer process is shown in Fig. 7;

```
do {
     . . .
   // produce an item in nextp
     . . .
   wait(empty);
   wait(mutex);
     . . .
   // add nextp to buffer
     . . .
   signal(mutex);
   signal(full);
}while (TRUE);
```

Figure: The structure of the producer process.

The Bounded-Buffer Problem II

The code for the consumer process is shown in Fig. 8;

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

Figure: The structure of the consumer process.

The Bounded-Buffer Problem III

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                      /* TRUE is the constant 1 */
        item = produce_item( );         /* generate something to put in buffer */
        down(&empty);                   /* decrement empty count */
        down(&mutex);                   /* enter critical region */
        insert_item(item);              /* put new item in buffer */
        up(&mutex);                     /* leave critical region */
        up(&full);                      /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                      /* infinite loop */
        down(&full);                    /* decrement full count */
        down(&mutex);                   /* enter critical region */
        item = remove_item( );          /* take item from buffer */
        up(&mutex);                     /* leave critical region */
        up(&empty);                     /* increment count of empty slots */
        consume_item(item);             /* do something with the item */
    }
}
```

Figure: The producer-consumer problem using semaphores.

The Readers-Writers Problem I

- A database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database (readers), whereas others may want to update (that is, to read and write) the database (writers).
- If two readers access the shared data simultaneously, no adverse affects will result.
- However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, there could be some **synchronization issues**.
- To ensure that these difficulties do not arise, we require that the writers have *exclusive access* to the shared database.
- This synchronization problem is referred to as the **readers-writers** problem.

The Readers-Writers Problem II

- The readers-writers problem has several variations, all involving priorities.
  - The simplest one, referred to as the first readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.
  - The second readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

The Readers-Writers Problem III

- A solution to either problem may result in **starvation**.
  - In the first case, writers may starve.
  - In the second case, readers may starve.
- The solution to the first readers-writers problem;

```
do {
   wait(wrt);
     . . .
   // writing is performed
     . . .
   signal(wrt);
}while (TRUE);
```

Figure: The structure of a writer process.

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
}while (TRUE);
```

Figure: The structure of a reader process.

The Readers-Writers Problem IV

- In the solution to the first readers-writers problem, the reader processes share the following data structures:

```
semaphore mutex, $wrt$;
int readcount;
```

  - The semaphores *mutex* and *wrt* are initialized to 1; *readcount* is initialized to 0.
  - The semaphore *wrt* is common to both reader and writer processes.
  - The *mutex* semaphore is used to ensure mutual exclusion when the variable *readcount* is updated.
  - The *readcount* variable keeps track of how many processes are currently reading the object.
  - The semaphore *wrt* functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the CS.
  - It is not used by readers who enter or exit while other readers are in their CSs.
- Note that, if a writer is in the CS and n readers are waiting, then one reader is queued on *wrt*, and $n - 1$ readers are queued on *mutex*.

The Dining-Philosophers Problem I

- The dining philosophers problem is useful for modeling processes that are *competing for exclusive access to a limited number of resources*, such as I/O devices.
- Consider five philosophers who spend their lives thinking and eating.
- The philosophers share a circular table surrounded by five chairs (see Fig. 12).

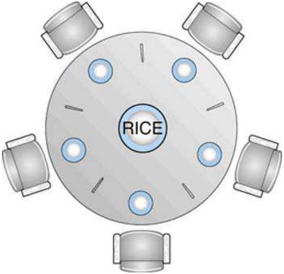

Figure: The situation of the dining philosophers.

The Dining-Philosophers Problem II

- The dining-philosophers problem is an example of a large class of concurrency-control problems.
  - When a philosopher thinks, she does not interact with her colleagues.
  - From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
  - A philosopher may pick up only one chopstick at a time.
  - When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
  - When she is finished eating, she puts down both of her chopsticks and starts thinking again.
- One simple solution is to represent each chopstick with a semaphore.
  - A philosopher tries to grab a chopstick by executing a *wait*() operation on that semaphore; she releases her chopsticks by executing the *signal*() operation on the appropriate semaphores.
  - Thus, the shared data are

    `semaphore chopstick[5] ;`

    where all the elements of chopstick are initialized to 1.

The Dining-Philosophers Problem III

- The structure of philosopher *i* is shown in Fig. 13.

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

Figure: The structure of philosopher *i*.

- Although this solution guarantees that no two neighbors are eating simultaneously, it could create a deadlock.
- Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick.
- When each philosopher tries to grab her right chopstick, she will be delayed forever.

The Dining-Philosophers Problem IV

- One improvement to Fig. 13 that has no deadlock and no starvation is to protect the five statements following the call to think by a binary semaphore.
  - Before starting to acquire forks, a philosopher would do a **down** on *mutex*
  - After replacing the forks, she would do an **up** on *mutex*
- It has a performance bug: only one philosopher can be eating at any instant.
- With five forks available, we should be able to allow two philosophers to eat at the same time.
- Any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death.

The Dining-Philosophers Problem V

- The solution presented in Fig. 14 is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers.
- It uses an array, *state*, to keep track of whether a philosopher is *eating*, *thinking*, or *hungry* (trying to acquire forks).
- A philosopher may move only into eating state if neither neighbor (LEFT and RIGHT) is eating .

```
#define N           5               /* number of philosophers */
#define LEFT        (i+N-1)%N       /* number of i's left neighbor */
#define RIGHT       (i+1)%N         /* number of i's right neighbor */
#define THINKING    0               /* philosopher is thinking */
#define HUNGRY      1               /* philosopher is trying to get forks */
#define EATING      2               /* philosopher is eating */
typedef int semaphore;              /* semaphores are a special kind of int */
int state[N];                       /* array to keep track of everyone's state */
semaphore mutex = 1;                /* mutual exclusion for critical regions */
semaphore s[N];                     /* one semaphore per philosopher */

void philosopher(int i)             /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {                  /* repeat forever */
        think( );                   /* philosopher is thinking */
        take_forks(i);              /* acquire two forks or block */
        eat( );                     /* yum-yum, spaghetti */
        put_forks(i);               /* put both forks back on table */
    }
}
```

Figure: A solution to the dining philosophers problem I.

The Dining-Philosophers Problem VI

```
void take_forks(int i)                 /* i: philosopher number, from 0 to N−1 */
{
     down(&mutex);                     /* enter critical region */
     state[i] = HUNGRY;                /* record fact that philosopher i is hungry */
     test(i);                          /* try to acquire 2 forks */
     up(&mutex);                       /* exit critical region */
     down(&s[i]);                      /* block if forks were not acquired */
}

void put_forks(i)                      /* i: philosopher number, from 0 to N−1 */
{
     down(&mutex);                     /* enter critical region */
     state[i] = THINKING;              /* philosopher has finished eating */
     test(LEFT);                       /* see if left neighbor can now eat */
     test(RIGHT);                      /* see if right neighbor can now eat */
     up(&mutex);                       /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N−1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```

Figure: A solution to the dining philosophers problem II.

The solution is deadlock–free and allows the maximum parallelism for any number of philosophers

Monitors I

- Although semaphores provide a convenient and effective mechanism for process synchronization,
- using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place and these sequences do not always occur.
- The semaphore solution to the CS problem.
  - All processes share a semaphore variable *mutex*, which is initialized to 1.
  - Each process must execute *wait*(*mutex*) before entering the CS and *signal*(*mutex*) afterward.
  - If this sequence is not observed,two processes may be in their CSs simultaneously.

Monitors II

- Suppose that a process interchanges the order in which the *wait*() and *signal*() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
    ...
 critical section
    ...
wait(mutex);
```

- Suppose that a process replaces *signal*(*mutex*) with *wait*(*mutex*). That is, it executes

```
wait(mutex);
    ...
 critical section
    ...
wait(mutex);
```

  In this case, a deadlock will occur.

- Suppose that a process omits the *wait*(*mutex*), or the *signal*(*mutex*), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

Monitors III

- You must be careful when using semaphores. It is like programming in assembly language, only worse, because the errors are race conditions, deadlocks, and other forms of unpredictable and irreproducible behavior.
- Semaphores require programmer to think of every timing issue; easy to miss something, difficult to debug.
- Let the compiler handle the details. Programmer only has to say what to protect.
- Researchers have developed **high-level language constructs - monitor**.
    - A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
    - Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.

Monitors IV

- Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant.
- Compiler actually does the protection (compiler will use semaphores to do protection).
- Main problem: provides less control.
- Some real programming languages also support monitors. One such language is Java.
- Java is an object-oriented language that supports user-level threads and also allows methods (procedures) to be grouped together into classes (keyword **synchronized**).

Usage I

- The monitor type contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.
- The syntax of a monitor is shown in Fig. 16.

```
monitor monitor name
{
  // shared variable declarations

  procedure P1 ( . . . ) {
     . . .
  }

  procedure P2 ( . . . ) {
     . . .
  }
          .
          .
          .
  procedure Pn ( . . . ) {
     . . .
  }

  initialization code ( . . . ) {
     . . .
  }
}
```

Figure: Syntax of a monitor.

Usage II

- The monitor construct ensures that only one process at a time can be active within the monitor.
- Consequently, the programmer does not need to code this synchronization constraint explicitly (see Fig. 17).

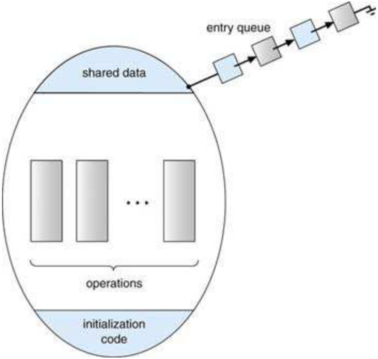

Figure: Schematic view of a monitor.