Process Synchronization – Outline

- Why do processes need synchronization ?
- What is the critical-section problem ?
- Describe solutions to the critical-section problem
  - Peterson’s solution
  - using synchronization hardware
  - semaphores
  - monitors
- Classic Problems of Synchronization

Why Process Synchronization ?

- Processes may *cooperate* with each other
  - producer-consumer and service-oriented system models
  - exploit concurrent execution on multiprocessors
- Cooperating processes may share data (globals, files, etc)
  - imperative to maintain data *correctness*
- Why is data correctness in danger ?
  - process run asynchronously, context switches can happen at any time
  - processes may run concurrently
  - different orders of updating shared data may produce different values
- Process synchronization
  - to coordinate updates to shared data
  - order of process execution should not leave shared data in an inconsistent state
- *Only needed when processes share data !*

Producer-Consumer Data Sharing

Producer

```
while (true){

    /* wait if buffer full */
    while (counter == 10)
     ;  /* do nothing */

    /* produce data */
    buffer[in] = pdata;
    in = (in + 1) % 10;

    /* update number of
        items in buffer */
    counter++;
}
```

Consumer

```
while (true){

    /* wait if buffer empty */
    while (counter == 0)
     ;  /* do nothing */

    /* consume data */
    cdata = buffer[out];
    out = (out + 1) % 10;

    /* update number of
        items in buffer */
    counter--;
}
```

Producer-Consumer Data Sharing

Producer

```
while (true){

    /* wait if buffer full */
    while (counter == 10)
     ;  /* do nothing */

    /* produce data */
    buffer[in] = sdata;
    in = (in + 1) % 10;

    /* update number of
        items in buffer */
    R1 = load (counter);
    R1 = R1 + 1;
    counter = store (R1);
}
```

Consumer

```
while (true){

    /* wait if buffer empty */
    while (counter == 0)
     ;  /* do nothing */

    /* consume data */
    sdata = buffer[out];
    out = (out + 1) % 10;

    /* update number of
        items in buffer */
    R2 = load (counter);
    R2 = R2 – 1;
    counter = store (R2);
}
```

Race Condition

- Suppose *counter* = *5*

Incorrect Sequence 1

```
R1 = load (counter);
R1 = R1 + 1;
R2 = load (counter);
R2 = R2 – 1;
counter = store (R1);
counter = store (R2);
```

<u>Final Value in counter = 4!</u>

Incorrect Sequence 2

```
R1 = load (counter);
R1 = R1 + 1;
R2 = load (counter);
R2 = R2 – 1;
counter = store (R2);
counter = store (R1);
```

<u>Final Value in counter = 6!</u>

- Race condition is a situation where
  - several processes concurrently manipulate shared data, and
  - final shared data value depends on the order of execution

Critical Section Problem

- Region of code in a process *updating* shared data is called a critical region.
- Concurrent updating of shared data by multiple processes is dangerous.
- Critical section problem
  - how to ensure synchronization between cooperating processes ?
- Solution to the critical section problem
  - only allow a single process to enter and be in its critical section at a time
- Protocol for solving the critical section problem
  - request permission to enter critical section
  - indicate after exit from critical section
  - only permit a single process at a time

Solution to the Critical Section Problem

- Formally states, each solution should ensure
  - *mutual exclusion:* only a single process can execute in *its* critical section at a time
  - *progress:* selection of a process to enter its critical section should be fair, and the decision cannot be postponed indefinitely.
  - *bounded waiting:* there should be a fixed bound on how long it takes for the system to grant a process's request to enter its critical section
- Other than satisfying these requirements, the system should also guard against *deadlocks.*

Preemptive Vs. Non-preemptive Kernels

- Several kernel processes share data
  - structures for maintaining file systems, memory allocation, interrupt handling, etc.
- How to ensure OSes are free from race conditions ?
- Non–preemptive kernels
  - process executing in kernel mode cannot be preempted
  - disable interrupts when process is in kernel mode
  - what about multiprocessor systems ?
- Preemptive kernels
  - process executing in kernel mode can be preempted
  - suitable for real-time programming
  - more responsive

Peterson's Solution to Critical Section Problem

- Software based solution
- Only supports two processes
- The two processes share two variables:
  - int turn;
    - indicates whose turn it is to enter the critical section
  - boolean flag[2]
    - indicates if a process is ready to enter its critical section

Peterson's Solution

Process 0

```
do {
   flag[0] = TRUE;
   turn = 1;
   while (flag[1]  &&  turn==1)
      ;
   // critical section

   flag[0] = FALSE;

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

Process 1

```
do {
   flag[1] = TRUE;
   turn = 0;
   while (flag[0]  &&  turn==0)
      ;
   // critical section

   flag[1] = FALSE;

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

- Solution meets all three requirements
  - P0 and P1 can never be in the critical section at the same time
  - if P0 does not want to enter critical region, P1 does no waiting
  - process waits for at most one turn of the other to progress

Peterson's Solution – Notes

- Only supports two processes
  - generalizing for more than two processes has been achieved
- Assumes that the LOAD and STORE instructions are atomic
- Assumes that memory accesses are not reordered
- May be less efficient than a hardware approach
  - particularly for >2 processes

Lock-Based Solutions

- General solution to the critical section problem
  - critical sections are protected by locks
  - process must acquire lock before entry
  - process releases lock on exit

```
do {
        acquire lock;

        critical section

        release lock;

        remainder section

} while(TRUE);
```

Hardware Support for Lock-Based Solutions – Uniprocessors

- For uniprocessor systems
  - concurrent processes cannot be overlapped, only *interleaved*
  - process runs until it invokes system call, or is *interrupted*
- Disable interrupts !
  - active process will run without preemption

```
do {

        disable interrupts;
        critical section
        enable interrupts;

        remainder section
} while(TRUE);
```

Hardware Support for Lock-Based Solutions – Multiprocessors

- In multiprocessors
  - several processes share memory
  - processors behave independently in a peer manner
- Disabling interrupt based solution will not work
  - too inefficient
  - OS using this not broadly scalable
- Provide hardware support in the form of atomic instructions
  - atomic *test-and-set* instruction
  - atomic *swap* instruction
  - atomic *compare-and-swap* instruction
- Atomic execution of a set of instructions means that instructions are treated as a single step that cannot be interrupted.

TestAndSet Instruction

- Pseudo code definition of *TestAndSet*

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

Mutual Exclusion using *TestAndSet*

```
int mutex;
init_lock (&mutex);

do {

    lock (&mutex);
        critical section
    unlock (&mutex);

        remainder section
} while(TRUE);
```

```
void init_lock (int *mutex)
{
    *mutex = 0;
}

void lock (int *mutex)
{
    while(TestAndSet(mutex))
        ;
}

void unlock (int *mutex)
{
    *mutex = 0;
}
```

Swap Instruction

- Psuedo code definition of swap instruction

```
void Swap (boolean *a, boolean *b)
 {
        boolean temp = *a;
        *a = *b;
        *b = temp:
 }
```

Mutual Exclusion using *Swap*

```
int mutex;
init_lock (&mutex);

do {

    lock (&mutex);
        critical section
    unlock (&mutex);

        remainder section
} while(TRUE);
```

```
void init_lock (int *mutex) {
    *mutex = 0;
}

void lock (int *mutex) {
    int key = TRUE;
    do {
        Swap(&key, mutex);
    }while(key == TRUE);
}

void unlock (int *mutex) {
    *mutex = 0;
}
```

- *Fairness not guaranteed by any implementation !*

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Cycle = 0

lock=FALSE, key=FALSE, waiting[0]=0, waiting[1]=0

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Cycle = 1

lock=FALSE, key=FALSE, waiting[0]=1, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Cycle = 2

lock=FALSE, key=TRUE, waiting[0]=1, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Cycle = 3

lock=FALSE, key=TRUE, waiting[0]=1, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Process 0 wins the race

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Cycle = 4

lock=TRUE, key=FALSE, waiting[0]=1, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Cycle = 5

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Cycle = 6

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

j = 1

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Cycle = 7

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Cycle = 8

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Cycle = 9

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=1

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Cycle = 10

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=0

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Cycle = 11

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=0

Bounded Waiting Solution

Process i = 0

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Process i = 1

```
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&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);
```

Cycle = 12

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=0