CISC 7310X
C09: Process Synchronization

Hui Chen

Department of Computer & Information Science

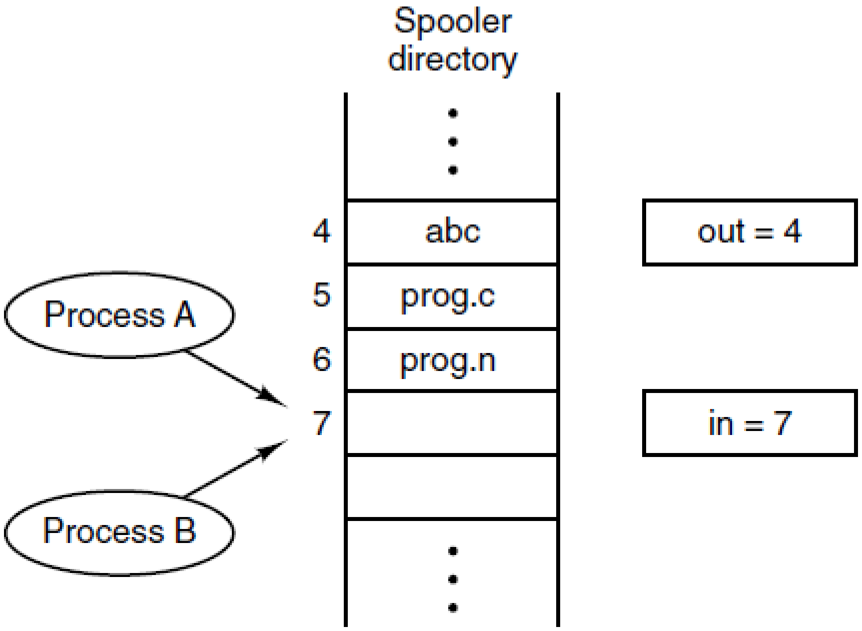
CUNY Brooklyn College

Outline

- Race condition and critical regions
  - The bounded buffer problem
- Process coordination
  - mutual exclusion
  - synchronization
- Examples

Race Condition

- Process A and B writes to the same slot



- [Figure 2-21 in Tanenbaum & Bos, 2014]

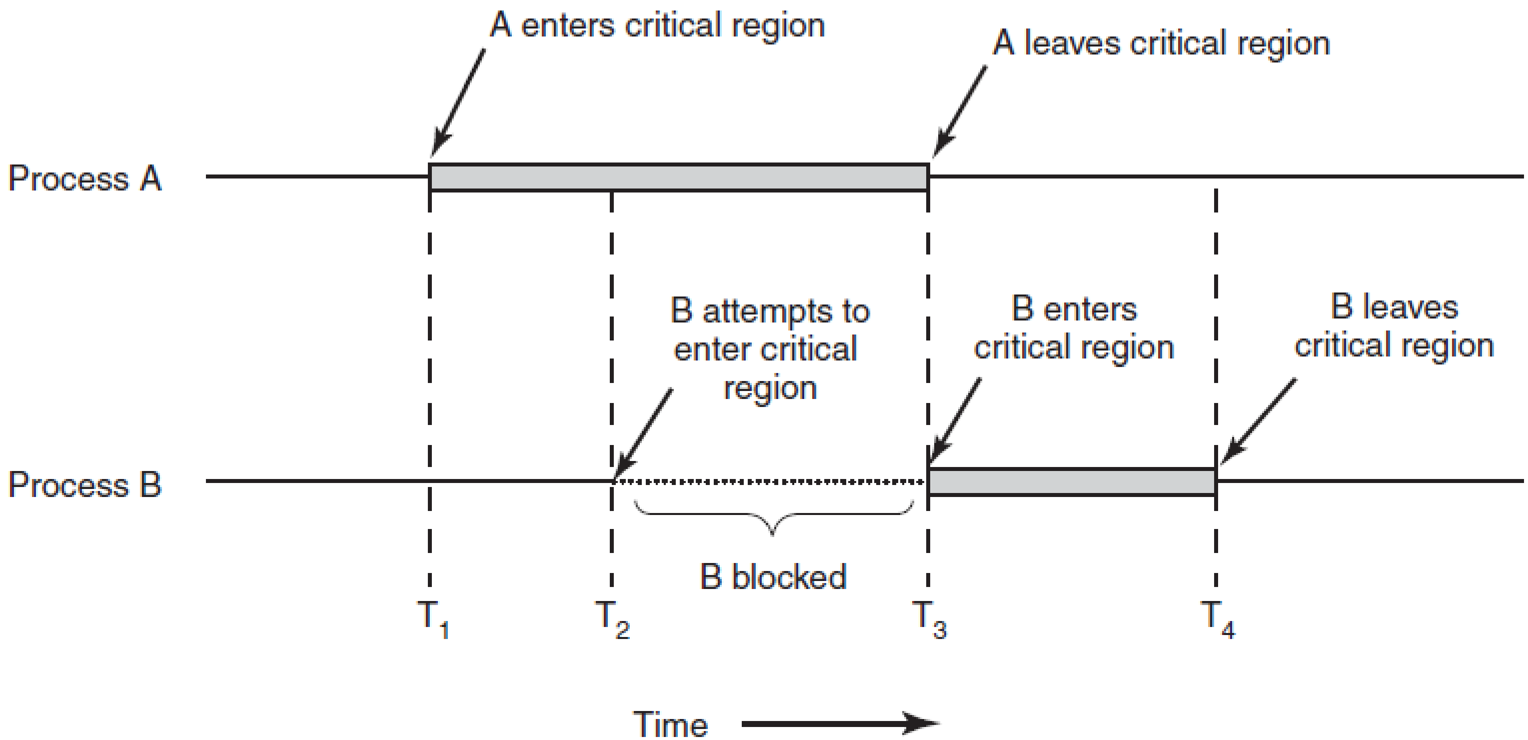
Mutual Exclusion and Critical Region

- Mutual exclusion
  - Disable allow multiple processes read and write at the same time to a shared resource
- Critical region (or critical section)
  - The part of the program where the shared memory is accessed.

Requirements to Avoid Race Conditions

1. No two processes may be simultaneously inside their critical regions.
2. No process running outside its critical region may block other processes.
3. No process should have to wait forever to enter its critical region.
4. No assumptions may be made about speeds or the number of CPUs.

Mutual Exclusion using Critical Section



- [Figure 2-22 in Tanenbaum & Bos, 2014]

Mutual Exclusion with Busy Waiting

- Disable interrupts
- Strict alternation
- Peterson's solution
- Synchronization hardware
  - TSL or XCHG instructions

Disable Interrupts

- Disable all interrupts just after entering critical section, and reenable them just before leaving
  - No clock interrupt occur, no switching process before read and write completes
- Applicable to only single processor single core system
- Generally, not appropriate to user programs
  - Otherwise all user programs must be placed in kernel mode

Strict Alternation

- Busy waiting or spin lock
  - Can a process be blocked by others while it is in non-critical section?

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)

- [Figure 2-23 in Tanenbaum & Bos, 2014]

Peterson's Algorithm

```
#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 2-24 in Tanenbaum & Bos, 2014]

Special instruction

- Instruction with exclusive access to memory bus
- TSL
- XCHG

TSL Instruction

- Test-Set-Lock instruction

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was nonzero, lock was set, so loop
        RET                        | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET                        | return to caller
```

- [Figure 2-25 in Tanenbaum & Bos, 2014]

XCHG Instruction

```
enter_region:
    MOVE REGISTER,#1          | put a 1 in the register
    XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
    CMP REGISTER,#0           | was lock zero?
    JNE enter_region          | if it was non zero, lock was set, so loop
    RET                       | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0              | store a 0 in lock
    RET                       | return to caller
```

- [Figure 2-26 in Tanenbaum & Bos, 2014]

Questions

- Busy-waiting solutions?

Non-Busy-Waiting Locking Approaches

- Producer-consumer problem with bounded buffer
- Sleep and wake-up
- Semaphores
  - Mutex
- Monitors
- Message Passing
- Barriers

Priority Inversion Problem

- Busy-waiting, waste CPU cycles
- Priority inversion problem
  - Assumptions
    - Two processes, H with high priority, L with low priority
    - Run H whenever it is in READY state
  - Consider sequence
    - L enters critical section
    - H becomes ready to run (which state is L in?)
    - H enters busy-waiting cycle
    - L never gets CPU cycle to leave its critical section, H loops forever in busy-waiting cycle

Producer-consumer problem

- Consider a bounded circular buffer shared by two processes
  - N slots
  - Producer adds item to the buffer, the item occupies a slot
  - Consumer consumes item from the buffer, free a slot
  - Counter counts items in the buffer
    - counter == 0, empty buffer, consumer must wait
    - counter == N, full buffer, producer must wait

Sleep and Wake-up

- Sleep
  - A system call that causes the caller to block (suspends the calling process, until another process wakes it up)
- Wakeup
  - A system call that wakes up another process

Producer

```
#define N 100                              /* number of slots in the buffer */
int count = 0;                             /* number of items in the buffer */

void producer(void)
{
        int item;

        while (TRUE) {                     /* repeat forever */
                item = produce_item( );    /* generate next item */
                if (count == N) sleep( );  /* if buffer is full, go to sleep */
                insert_item(item);         /* put item in buffer */
                count = count + 1;         /* increment count of items in buffer */
                if (count == 1) wakeup(consumer); /* was buffer empty? */
        }
}


void consumer(void)
{
```

Consumer

```
            if (count == 1) wakeup(consumer);            /* was buffer empty? */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                               /* repeat forever */
                if (count == 0) sleep( );            /* if buffer is empty, got to sleep */
                item = remove_item( );               /* take item out of buffer */
                count = count − 1;                   /* decrement count of items in buffer */
                if (count == N − 1) wakeup(producer); /* was buffer full? */
                consume_item(item);                  /* print item */
        }
}
```

- [Figure 2-27 in Tanenbaum & Bos, 2014]

Can a race occur?

- Depends on checking condition
  - Example
    - Producer: if (count == N) sleep();
    - Consumer: if (counter ==0) sleep();
  - How about
    - Producer: counter = counter + 1;
    - Consumer: counter = counter – 1;
    - How many instructions needed to compute the above?

Semaphores

- A variable with two atomic operations

```
Down( semaphore *V) {
        if (V->value > 0) V->value --;
        else if (V->value == 0) { add this process to S->list; Sleep(); // remains in "Down" before waking up }
}
Up (semaphore *V) {
        V->value ++;
        if (V->list is not empty) { remove a process P from V->list; Wake(P); }
}
```

- These operations are designed to be atomic with the help of
  - Either interrupts (single processor & single core)
  - Or TSL or XCHG instruction to implement a spin locks (on multi-processor or multi-core system)

Producer-Consumer: Semaphore

- In total, three semaphores, producer uses two(empty & mutex), and consumer also two (full & mutex)
  - empty: whether buffer is empty, if empty, consumer should sleep
  - full: whether buffer is full, if full, producer should sleep
  - mutex: whether other is in critical section, if so, sleep
- empty & full are for synchronization (coordination)
- mutex is for achieving mutual exclusion

Producer with Two Semaphores

```
#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)
```

- [Figure 2-28 in Tanenbaum & Bos, 2014]

Consumer with Two Semaphores

```
            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 2-28 in Tanenbaum & Bos, 2014]

Mutexes (Mutex Locks)

- When counting is not needed, we semaphore can be made simpler
  - A semaphore with two states
    - Locked (corresponding to down)
      - Locked processes are sleeping
    - Unlocked (corresponding to up)
      - Unlocking a process is to wake up a sleeping process

Mutex: Implementation

```
mutex_lock:
        TSL REGISTER,MUTEX        | copy mutex to register and set mutex to 1
        CMP REGISTER,#0           | was mutex zero?
        JZE ok                    | if it was zero, mutex was unlocked, so return
        CALL thread_yield         | mutex is busy; schedule another thread
        JMP mutex_lock            | try again
ok:     RET                       | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0             | store a 0 in mutex
        RET                       | return to caller
```

- [Figure 2-29 in Tanenbaum & Bos, 2014]

Mutex in Pthreads

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

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

- [Figures 2-30, 2-31 in Tanenbaum & Bos, 2014]

Producer-Consumer: PThreads

- A Simple implementation in PThreads
- What would you comment about it? How is it easy to make mistake?

PThread: Producer

```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;                    /* used for signaling */
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);
}
```

- [Figures 2-32 in Tanenbaum & Bos, 2014]

PThread: Consumer

```
        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);
}

int main(int argc, char **argv)
```

- [Figures 2-32 in Tanenbaum & Bos, 2014]

PThread: Creating Threads

```
        pthread_exit(0);
}

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);
}
```

- [Figures 2-32 in Tanenbaum & Bos, 2014]

Questions?

- Priority inversion problem
- Producer-consumer problem
- Semaphore
  - Mutex lock
- PThread example

Monitor: Concept

- Semaphore
  - In practice, easily use it wrongly
    - Timing error
- Monitor: a high-level language construct that helps achieve mutual exclusion
  - Build on top of semaphore
  - Ensures that only process at a time is active within the monitor

Monitors

- Producer and consumer are mutually exclusive

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;


        procedure consumer( );
        .       .       .
        end;
end monitor;
```

- [Figures 2-33 in Tanenbaum & Bos, 2014]

Monitor: Producer-Consumer

```
monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count − 1;
        if count = N − 1 then signal(full)
    end;

    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

- [Figures 2-34 in Tanenbaum & Bos, 2014]

Monitor: In Java

- Java realizes monitor using synchronized methods

Message Passing

- Monitor does not work well for distributed systems
- Message passing
  - send(destination, &message)
  - receive(source, &message)
    - Can be blocking

Message Passing: Producer

```
#define N 100                              /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                            /* message buffer */

     while (TRUE) {
          item = produce_item( );          /* generate something to put in buffer */
          receive(consumer, &m);           /* wait for an empty to arrive */
          build_message(&m, item);         /* construct a message to send */
          send(consumer, &m);              /* send item to consumer */
     }
}

void consumer(void)
```

- [Figures 2-36 in Tanenbaum & Bos, 2014]

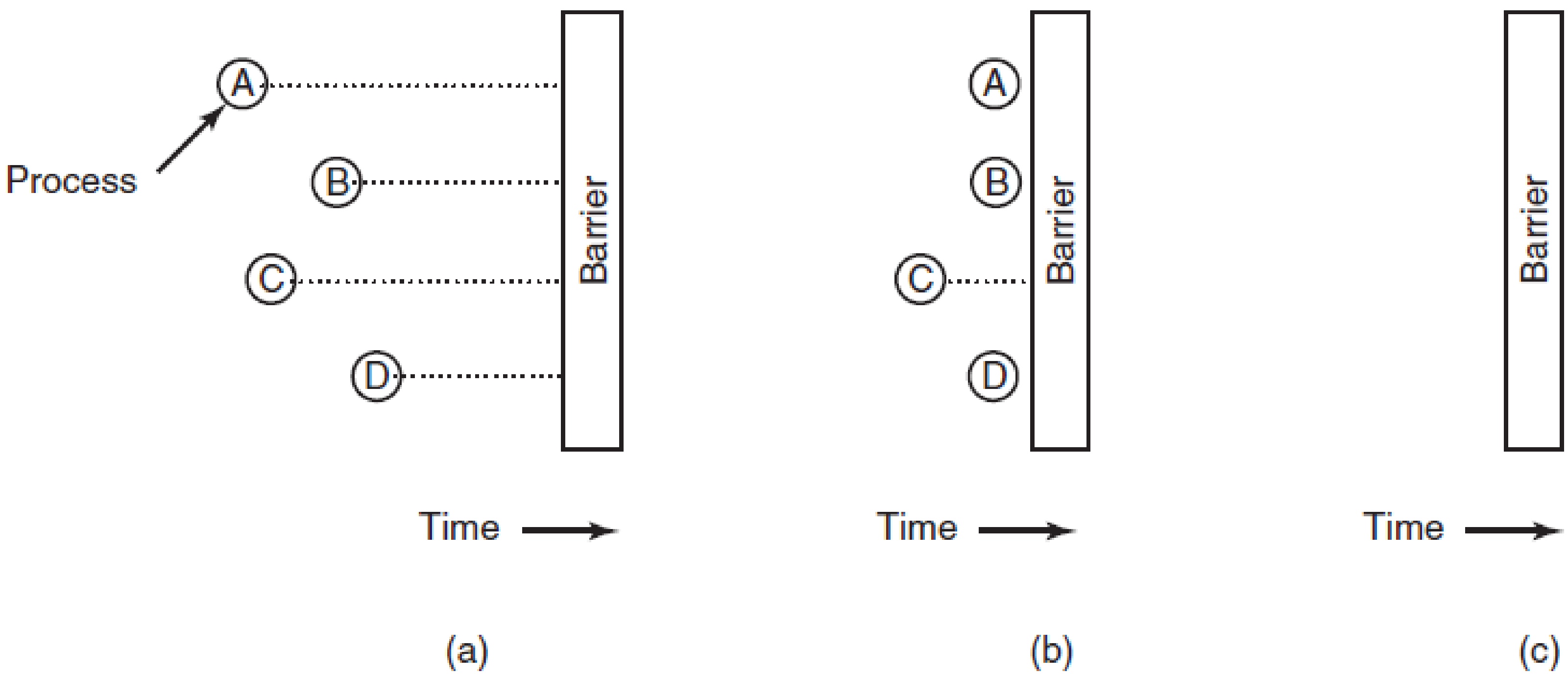
Message Passing: Consumer

```
            send(consumer, &m);                  /* send item to consumer */
        }
}

void consumer(void)
{
        int item, i;
        message m;

        for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
        while (TRUE) {
                receive(producer, &m);             /* get message containing item */
                item = extract_item(&m);           /* extract item from message */
                send(producer, &m);                /* send back empty reply */
                consume_item(item);                /* do something with the item */
        }
}
```

- [Figures 2-36 in Tanenbaum & Bos, 2014]

Barriers

- A synchronization mechanism



- [Figures 2-37 in Tanenbaum & Bos, 2014]

Questions

- Message passing and barriers

Assignment

- Practice assignment