Lecture 08 Concurrency and Synchronization

Dr. Tushar, Mosaddek Hossain Kamal
Professor

Computer Science and Engineering, University of Dhaka,

BSc Third Year, Semester 2 (July – Dec), Academic Year: 2022

CSE3201: Operating Systems

November 21, 2024

Outline

1. Learning Outcome
2. Concurrency and Synchronization
3. Test-and-Set
    - Producer-Consumer Problem using Test-and-Set
4. Semaphores
    - Producer-Consumer Problem with Semaphore
5. Monitors
    - Monitor example
    - Producer-Consumer using monitors
6. Example: Classical Synchronization problems

Learning Outcome

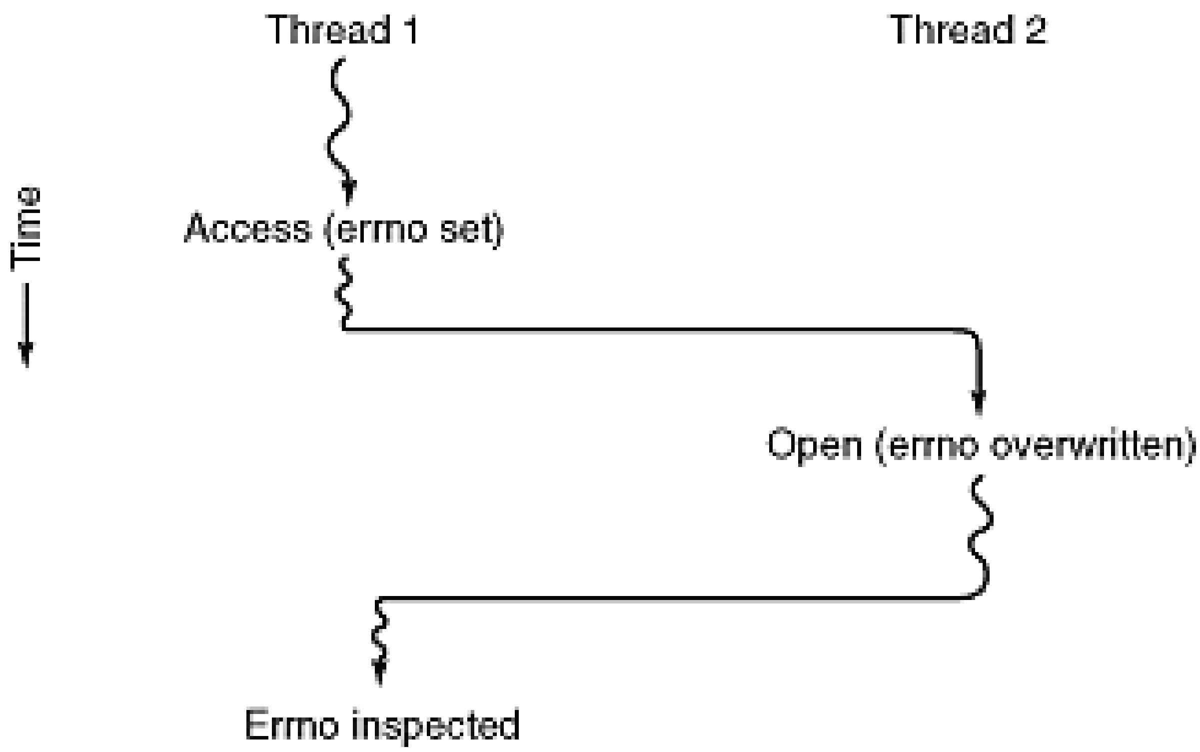
Learning Outcome

- **Understand the Operating System Process**
  - Process and Thread
  - Process and Thread context, data Structure
  - PCB and TCB
- **Understand the Process Management**
  - Context Switch and Dispatcher
  - Process Scheduling
  - Process and Thread Synchronization tools
  - Deadlock
- **Realtime process**
  - Concept of Soft ane Hard real time process
  - Reatime Process Scheduling

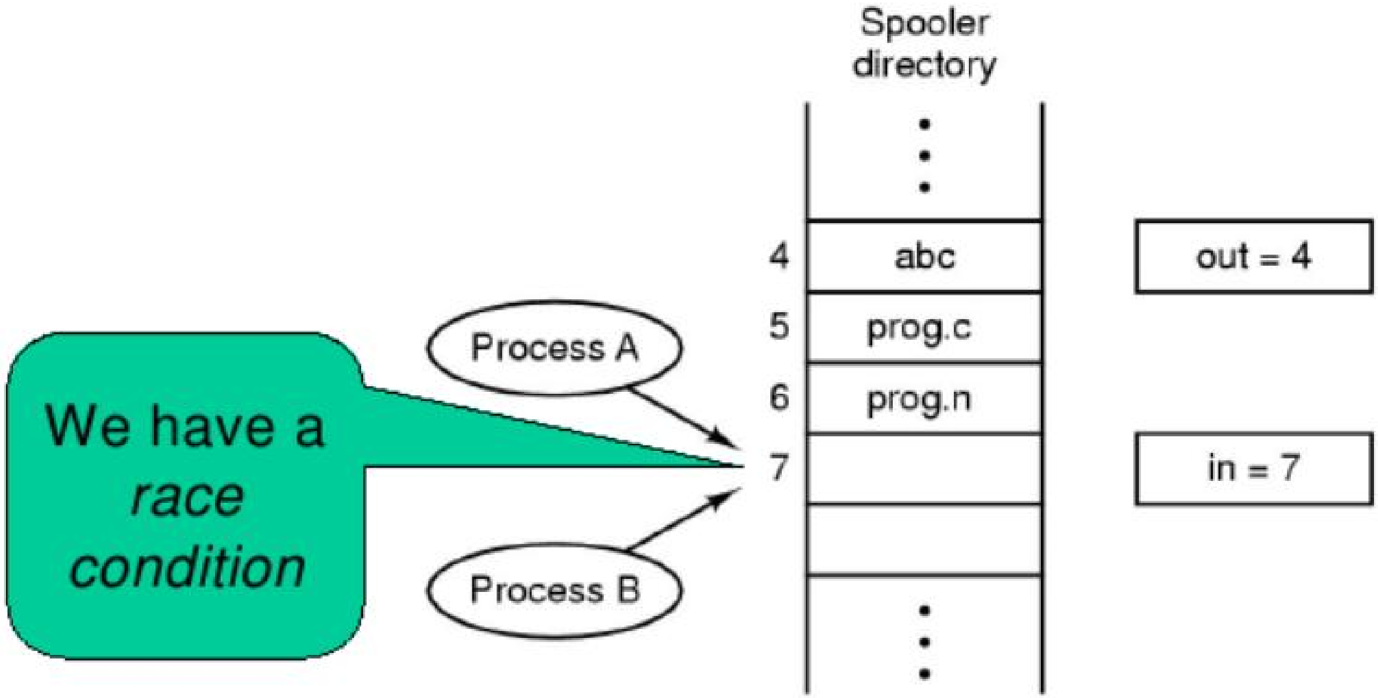
Concurrency and Synchronization

- Inter thread/Process communication
- Critical Region
- Solution
- Examples of IPC/ITC problems
- Tools used to overcome these IPC/ITC problems
  - Semaphore
  - Monitor
  - locks
- OS Synchronization Primitives (DUOS) – Cortex-M4
  - Lock
  - Semaphore
  - Condition variables

Making Single-Threaded Code Multithreaded



Conflict between threads over the use of a global variable



Two processes want to access shared memory at same time

Critical Region

- We can control access to the shared resources by controlling access to the code that accesses the resouce

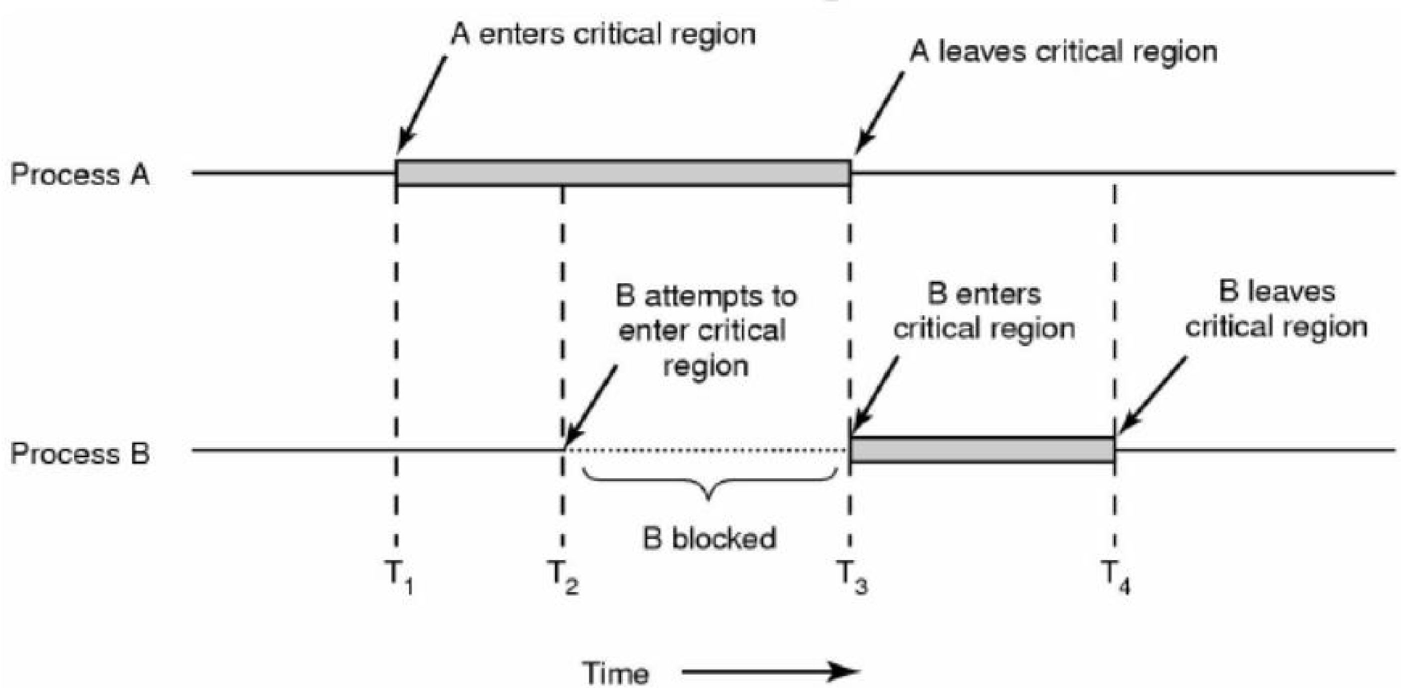
Critical Region

A critical region is a region of code where shared resources are accessed

- Variables, memory, files, etc.

- Unconditional entry to the critical region results in race condition
  - Incorrect behaviour, deadlock, lost work, ...

Critical Region



Mutual exclution using critical regions

Example Critical Region

```
struct node {
        int data;
        struct node *next;
};
struct node *head;
void init(void)
{
        head = NULL;
}
```

Simple first-in-first-out queue implemetation as a linked list Critical Region

```
void insert(struct *item)
{
item->next=head;
head = item;
}
struct node *remove(void)
{
 struct node *t;
 t = head;
 if(t!=NULL){
 head = head->next;
 }
return t;
}
```

Critical Region

Also known as critical section.
Conditions required of any solution to the critical region problem

- Mutual Exclusion:
  - No two processes simultaneously in critical region
- No assumptions made about speeds or numbers of CPUs
- Progress
  - No process running outside its critical region may block another process
- Bounded
  - No process must wait forever to enter its critical region

A solution?

- A lock variable
  - if lock == 1,
    - Somebody is in the critical section and we must wait
  - if lock == 0
    - nobody is in the critical section and we are free to enter

Solution

Process-0

```
while(TRUE) {
        while(lock ==1);
        lock=1;
        critical();
        lock=0;
        non_critical();
}
```

Process-1

```
while(TRUE) {
        while(lock ==1);
        lock=1;
        critical();
        lock=0;
        non_critical();
}
```

Problematic Execution Sequence

```
while(TRUE) {
                                    while(TRUE) {
                                      while(lock == 1);
  while(lock == 1);
  lock = 1;
                                      lock = 1;
  critical();                         critical();
  lock = 0
  non_critical();
}
                                      lock = 0
                                      non_critical();
                                    }
```

Observation

- Unfortunately, it is usually easier to show something does not work, than it is to prove that it does work.
  - Ideally, we'd like to prove, or at least informally demonstrate, that our solutions work.

Mutual Exclusion by Taking Turns

```
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)

Proposed solution to critical region problem (a) Process-0, (b) Process-1

Mutual Exclusion by Taking Turns

Works due to strict alternation

- Each process takes turns

Cons

- Busy waiting
- Process must wait its turn even while the other process is doing something else.
  - with many processes, must wait for everyone to have a turn
    - Does not guarantee progress if a process no longer needs a turn
  - Poor solution when processes require the critical section at differing rates

Mutual Exclusion by Disabling Interrupts

- Before entering a critical region, disable interrupts
- After leaving the critical region, enable interrupts
- Pros
  - simple
- Cons
  - Only available in the kernel
  - Blocks everybody else, even with no contention
    - Slows interrupt response time
  - Does not work on a multiprocessor

Hardware Support for mutual exclusion

Test and set instruction

- Can be used to implement lock variables correctly
  - it loads the value of the lock
  - if lock == 0
    - set the lock to 1
    - return result '0' – we acquire the lock
  - if lock == 1
    - return 1 – another thread/process has the lock
- Hardware guarantees that the instruction executes atomically
  - Atomically: As an indivisible unit.

Mutual Exclusion with Test-and-Set

```
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 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 oaller
```

Entering and leaving a critical region using the TSL instruction

Test-and-Set

Pros

- Simple (easy to show its correct)
- Available at user-level
  - To any number of processors
  - To implement any number of lock variables

Cons

- Busy waits (also termed a spin lock)
  - Consumes CPU
  - Livelock in the presence of priorities
    - If a low priority process has the lock and a high priority process attempts to get it, the high priority process will busy-wait forever.
  - Starvation is possible when a process leaves its critical section and more than one process is waiting.

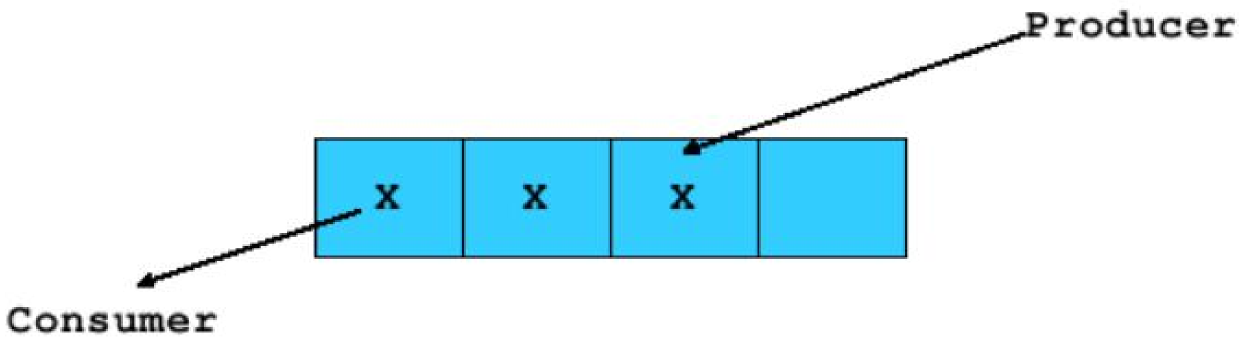
Tackling the Busy-Wait Problem

Busy Waiting?? What can we do?

- Sleep / Wakeup
  - The idea when process is waiting for an event, it calls sleep to block. instead of busy waiting.
  - The the event happens, the event generator (another process) calls wakeup to unblock the sleeping process.

The Producer-Consumer Problem

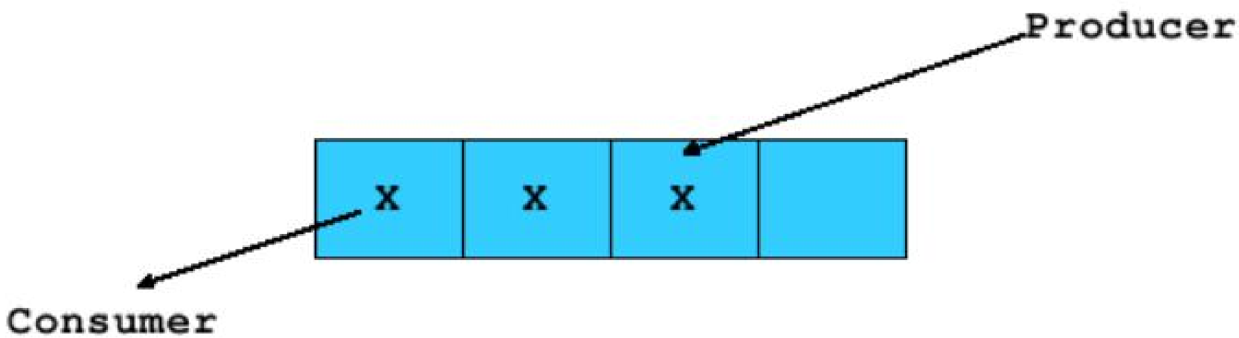
- Also called the bounded buffer problem
- A producer produces data items and stores the items in a buffer
- A consumer takes the items out of the buffer and consumes them.



Issues

We must keep an accurate count of items in buffer

- Producer
  - can sleep when the butter is full.
  - and wakeup when there is empty space in the buffer
    - The consumer can call wakeup when it consumes the first entry of the full buffer
- Consumer
  - Can sleep when the buffer is empty
  - And wake up when there are items available
    - Producer can call wakeup when it acids the first item to the butter



Pseudo-code for producer and consumer

Producer

```
count = 0;
#define N 4 /* buf size */
prod(){
 while(TRUE) {
        item = produce()
        if(count == N)
                sleep();
        insert_item();
        count++;
        if(count == 1)
            wakeup(con);
        }
}
```

Consumer

```
con(){
        while(TRUE){
        if(count == 0)
                sleep();
        remove_item();
        count++;
        if(count == N-1)
                wakeup();
        }
}
```

Problems

```c
int count = 0;
#define N 4 /* buf size */
prod() {
   while(TRUE) {
         item = produce()
         if (count == N)
               sleep();
         insert_item();
         count++;
         if (count == 1)
               wakeup(con);
   }
}
```

```c
con() {
   while(TRUE) {
         if (count == 0)
               sleep();
         remove_item();
         count--;
         if (count == N-1)
               wakeup(prod);
   }
}
```

Concurrent uncontrolled access to the buffer

Problems

```
int count = 0;
#define N 4 /* buf size */
prod() {
   while(TRUE) {
        item = produce()
        if (count == N)
                sleep();
        insert_item();
        count++;
        if (count == 1)
                wakeup(con);
   }
}
```

```
con() {
   while(TRUE) {
        if (count == 0)
                sleep();
        remove_item();
        count--;
        if (count == N-1)
                wakeup(prod);
}
```

Concurrent uncontrolled access to the counter

Proposed Solution

Lets use a locking primitive based on test-and-set to protect the concurrent access

```
int count = 0;
#define N 4 /* buf size */
prod() {
   while(TRUE) {
         item = produce()
         if (count == N)
                 sleep();
         acquire_lock()
         insert_item();
         count++;
         release_lock()
         if (count == 1)
                 wakeup(con);
   }
}
```

```
con() {
   while(TRUE) {
         if (count == 0)
                 sleep();
         acquire_lock()
         remove_item();
         count--;
         release_lock();
         if (count == N-1)
                 wakeup(prod);
   }
}
```

Problematic execution sequence

```
con() {
    while(TRUE) {
        if (count == 0)
```

```
prod() {
    while(TRUE) {
        item = produce()
        if (count == N)
            sleep();
        acquire_lock()
        insert_item();
        count++;
        release_lock()
        if (count == 1)
            wakeup(con);
```

wakeup without a matching sleep is lost

```
            sleep();
        acquire_lock()
        remove_item();
        count--;
        release_lock();
        if (count == N-1)
            wakeup(prod);
    }
```

Problem

- The test for *some condition* and actually going to sleep needs to be atomic
- the following does not work

```
acquire_lock();
if (count == N)
        sleep();
release_lock();
```

- The look is held while asleep $\Rightarrow$ count will never change

Semaphores

- **Dijkstra (1965) introduced two primitives that are more powerful than simple sleep and wakeup alone.**
  - P(): proberen, from Dutch to *test*.
  - V(): verhogen, from Dutch to *increment*.
  - Also called wait and signal , down & up

How it works?

- If a resource is not available, the corresponding semaphore blocks any process waiting for the resource
- Blocked processes are put into a process queue maintained by the semaphore (avoids busy waiting!)
- when a process releases a resource, it signals this by means of the semaphore
- Signalling resumes a blocked process if there is any
- Wait and signal operations cannot be interrupted
- Complex coordination can be implemented by multiple semaphores

Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {
        int count;
        struct process *L;  //PCB/TCB listed
}semaphore;
```

- Assume two simple operations:
  - **sleep** suspends the process that invokes it.
  - **wakeup(P)** resumes the execution of a blocked process P.

Semaphore operations now defined as

```
wait(S):
        S.count--;
        if(S.count<0){
                add this process to S.L;
                sleep
        }
signal(S):
        S.count++;
        if(S.count<=0){
                remove a process P from S.L;
                wakeup(P);
        }
```

- Each primive is atomic

Semaphore as a general sechronization tools

- Execute B in P_j only after A Executed in P_i
- Use semaphore *count* initialized to 0
- code:

$$\begin{array}{cc} P_i & P_j \\ \vdots & \vdots \\ A & wait(flag) \\ signal(flag) & B \end{array}$$

Semaphore implementation of a Mutex

- Mutex also called for Mutual Exclusion
  - Can also be called a lock

```
semaphore mutex;

mutex.count = 1; /*initialize mutex */

wait(mutex); /*before enter the critical region */

BlahBlah ..

signal(mutex); /* exit the critical region */
```

Notice that the initial count determines how many waits can progress before blocking and requiring a signal $\Rightarrow$ mutex.count initialized as 1

Solving the producer-consumer problem with semaphores

```
#define N = 4

semaphore mutex = 1;

/* count empty slots */
semaphore empty = N;

/* count full slots */
semaphore full = 0;
```

Solving the producer-consumer problem with semaphore

```
prod() {
 while(TRUE) {
        item = procedure();
        wait(empty);
        wait(mutex);
        insert_item();
        signal (mutex);
        signal(full);
 }
}
```

```
con() {
 while(TRUE) {
        wait(full);
        wait(mutex);
        remove_item;
        signal (mutex);
        signal(empty);
 }
}
```

Summarizing Semaphore

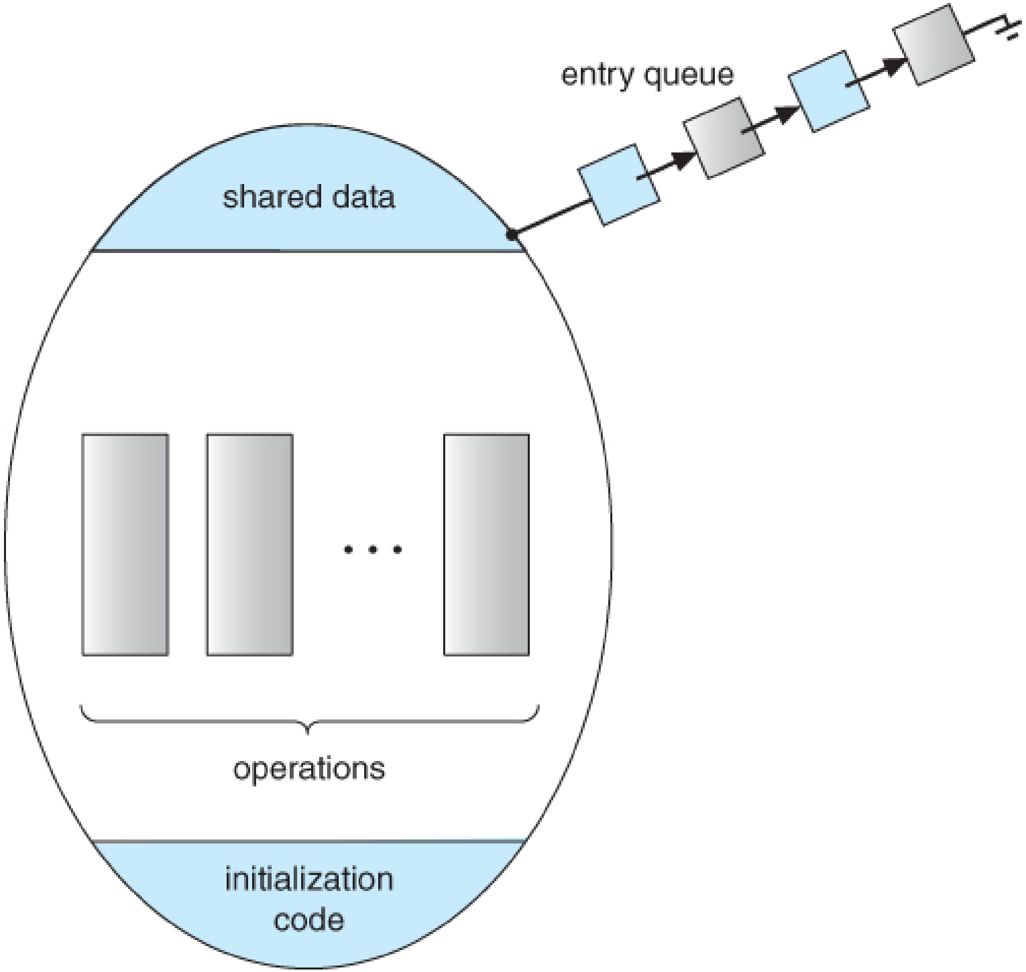
- Semaphore can beused to solve a variety of concurrency problems
- However, programming with semaphore can be error-prone
  - E.g. must signal for every wait for mutexes
    - Too many, or too few signals or waits, or signals and waits in the wrong order, can have catastrophic results

Monitors

- To ease the concurrency programming, Hoare (1974) proposed *monitors*
  - A higher level synchronization primitive
  - Programing language construct
- Idea
  - A set of procedure, variables, data types are grouped in a special kind of module, a monitor.
    - Variable and data types are only accessed from within the monitor
  - Only one process/thread can be in the monitor at any one time
    - Mutual excution is implemented by the compiler (which should be less error-prone)

Monitor

- When a thread calls a monitor procedure that has a thread already inside, it is queued and it sleeps until the current thread exits the monitor



Monitors

Example of a monitor:

```
monitor example
        integer i;
        condition c;

        procedure producer();
        .
        .
        .
        end;

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

Simple Example

```
monitor counter {
 int count;
 procedure inc(){
        count = count + 1;
 }
 procedure dec(){
        count = count - 1;
 }
}
```

Note: “paper” languge

- Compile guarantees only one thread an be active in the monitor at any one time
- Easy to see this provides mutual exclusion
  - No race condition

Monitor

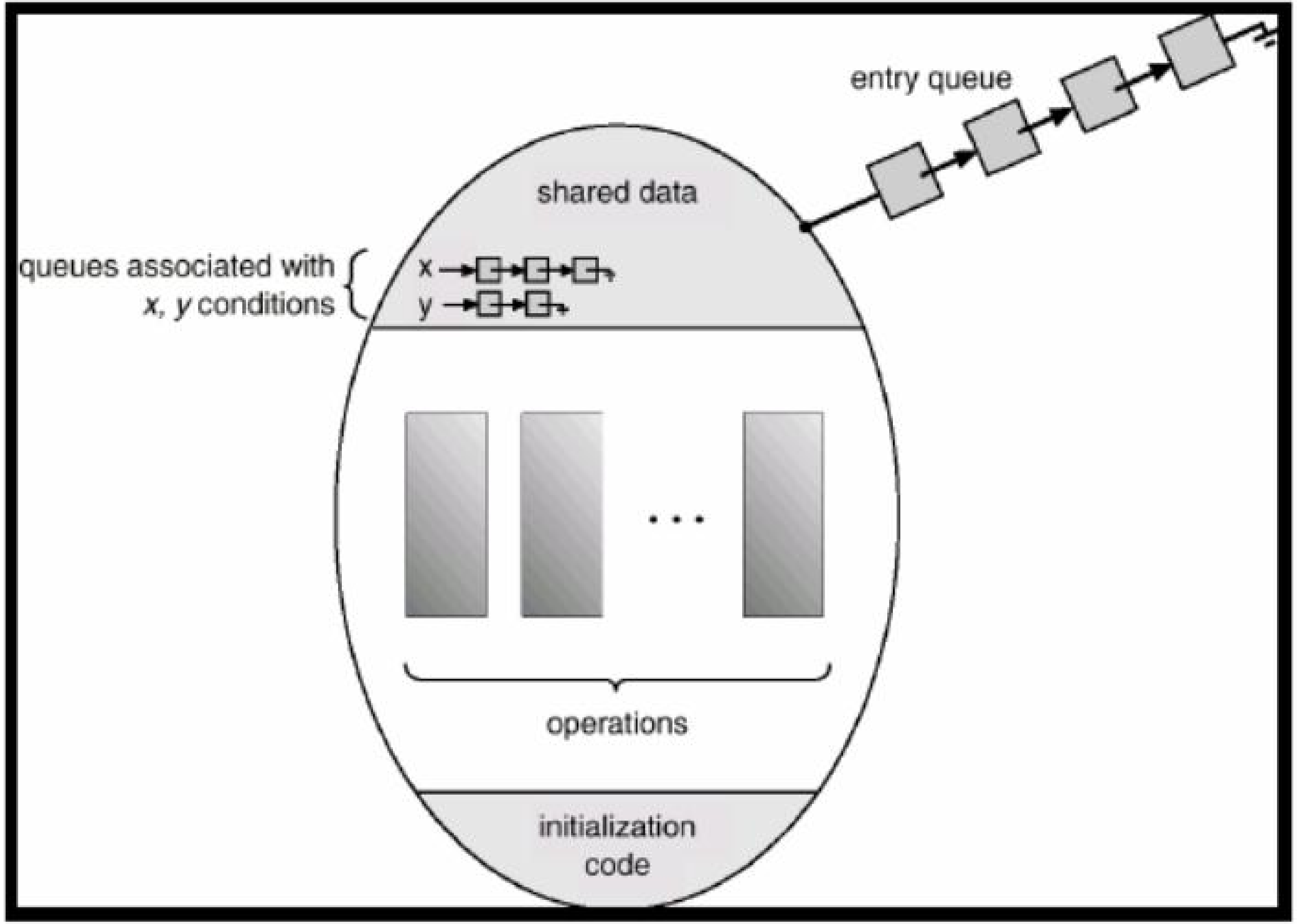
How do we block waiting for a event?

- We need a mechanism to block waiting for an event (in addition to ensure mutual exclusion)
  - e.g., for producer consumer problem when buffer is empty or full
- *Conditon Variables*

Condition Variable

- To allow a process to wait within the monitor, a **condition** variable must be declared as
  condition x, y
- Condition variable can only be used with the operations **wait** and **signal**
  - The operation
    x.wait();
    means that the process invoking theis operation is suspended until another process invokes
    x.signal();
  - The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

Condition Variables



Monitors

```
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;
```

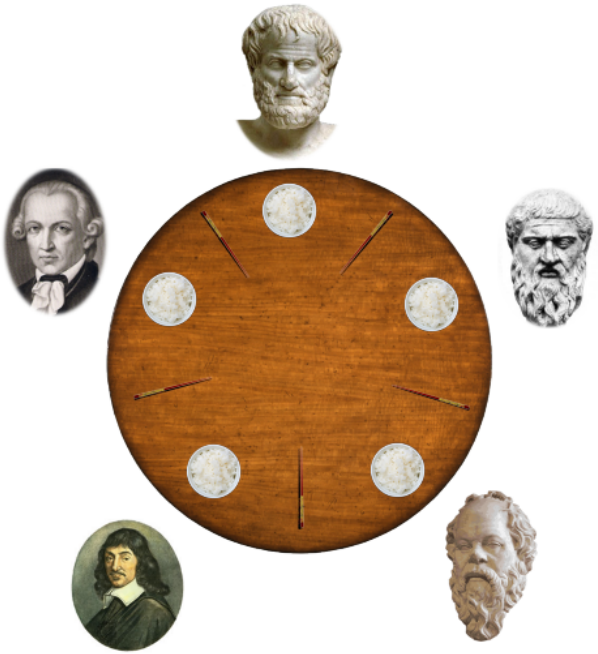
Outline of producer-consumer problem with monitors

- only one monitor procedure active at one time
- buffer has N slots

Example: Some Classical Problems

Dining Philosophers Problem

- Philosopher eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock



One Dining-Philosopher problem Solution

```
#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 */
    }
}
```

Solution to dining philosophers problem (part 1)

One Dining-Philosopher problem Solution contd.

```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                  /* philosopher is thinking */
          take_fork(i);              /* take left fork */
          take_fork((i+1) % N);      /* take right fork; % is modulo operator */
          eat( );                    /* yum-yum, spaghetti */
          put_fork(i);               /* put left fork back on the table */
          put_fork((i+1) % N);       /* put right fork back on the table */
     }
}
```

A nonsolution to dining philosophers problem

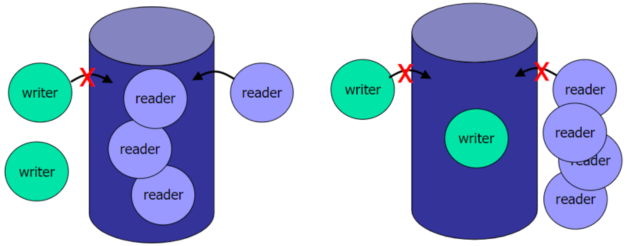
Dining Philosopher Problem Solution

```
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]);
      }
}
```

Solution to dining philosophers problem (part 2)

Readers and Writers Problem

- Models access to a database
  - E.g., airline reservation system
- Can have more than one consurrent reader
  - To check schedules and reservations
- Writers must have exclusive access
  - To book a ticket or update a schedule



Readers and Writers Problem

```
typedef int semaphore;              /* use your imagination */
semaphore mutex = 1;                /* controls access to 'rc' */
semaphore db = 1;                   /* controls access to the database */
int rc = 0;                         /* # of processes reading or wanting to */

void reader(void)
{
      while (TRUE) {                /* repeat forever */
            down(&mutex);           /* get exclusive access to 'rc' */
            rc = rc + 1;            /* one reader more now */
            if (rc == 1) down(&db); /* if this is the first reader ... */
            up(&mutex);             /* release exclusive access to 'rc' */
            read_data_base( );      /* access the data */
            down(&mutex);           /* get exclusive access to 'rc' */
            rc = rc − 1;            /* one reader fewer now */
            if (rc == 0) up(&db);   /* if this is the last reader ... */
            up(&mutex);             /* release exclusive access to 'rc' */
            use_data_read( );       /* noncritical region */
      }
}


void writer(void)
{
      while (TRUE) {                /* repeat forever */
            think_up_data( );       /* noncritical region */
            down(&db);              /* get exclusive access */
            write_data_base( );     /* update the data */
            up(&db);                /* release exclusive access */
      }
}
```

A solution to the readers and writers probllem

Sleeping Barber Problem

Figure 1: Sleeping Barber Problem

Sleeping Barber Problem

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting - 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair( );                /* cut hair (outside critical region) */
    }
}

void cu

{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut( );             /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

See the textbook