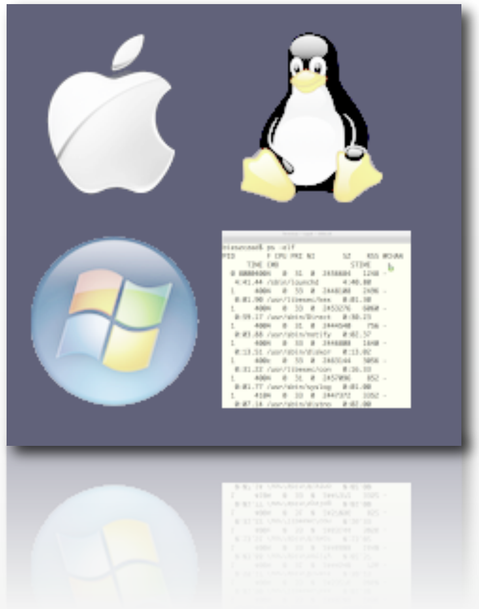


Lecture 7: Process Synchronization

COMP362 Operating Systems
Prof. AJ Bieszczad

Outline: Process Synchronization

- Background
- The Critical Section Problem
- Peterson's Software Solution to the Critical Section Problem
- Data Synchronization Challenges on Multiprocessor Systems
- Hardware Support for Synchronization
- The Busy Wait Problem
- Semaphores
- Monitors
- Condition Variables
- Classic Problems of Synchronization

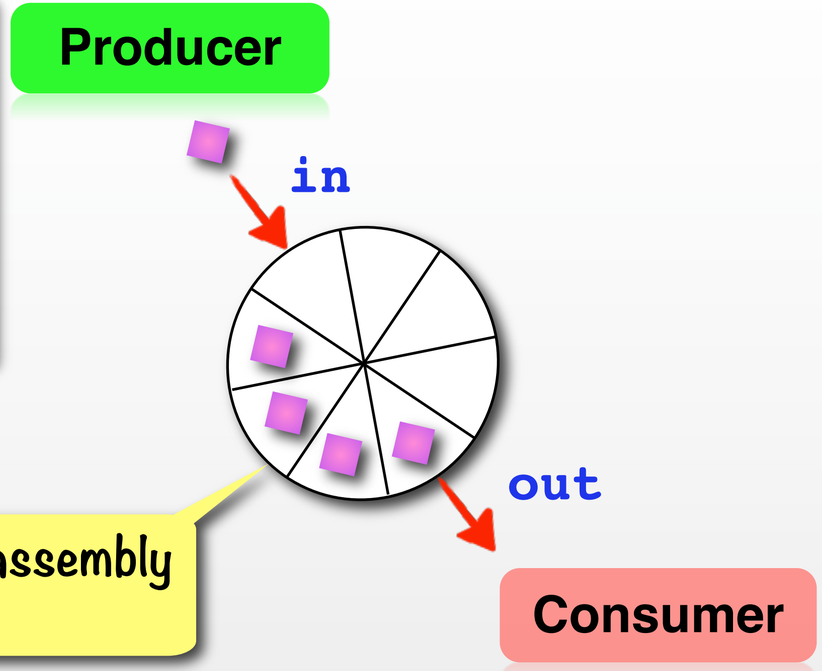
Background

- Multitasking exposes data to many processes/threads
- **Nondeterministic concurrent access** to **shared data** may result in **data inconsistency**
- Maintaining data consistency requires mechanisms to ensure orderly execution of cooperating processes

- Recall that in the producer-consumer paradigm, the producer writes to a circular buffer and the consumer reads from the same buffer
  - Suppose that we want to pass items from the producer to the consumer by using available buffer slots in a circular fashion
    - you may also think about it as a number of buffers rather than one buffer with multiple slots
  - We can do so by having an integer **`count`** that keeps track of the number of full slots (or buffers if you prefer)
    - Initially, **`count`** is set to 0. It is incremented by the producer after it deposits a new item in an available buffer slot
    - **`count`** is decremented by the consumer after it consumes a buffer

Producer/Consumer

```
while (true) {
// produce an item and put in nextProduced
   while (count == BUFFER_SIZE)
      ; // do nothing (i.e., wait for a slot)
   buffer [in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
   count++;
}
```



analogous to an assembly line in a factory

```
while (true)  {
   while (count == 0)
      ; // do nothing (i.e., wait for something)
   nextConsumed =  buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   count--;
   //  consume the item in nextConsumed
}
```

Process Races

- `count++` could be implemented as

```
register1 = count           // LOAD
register1 = register1 + 1   // INCREMENT
count = register1           // STORE
```

- `count--` could be implemented as

```
register2 = count           // LOAD
register2 = register2 - 1   // DECREMENT
count = register2           // STORE
```

- Assume (arbitrarily) that we are at state S0 and that `count = 5`
- The following is one possible sequence of context switches between producer and consumer:

one way the scheduler may order process execution

context switch

context switch

context switch

```
S0: producer executes LOAD register1, count    {register1 = 5}
S1: producer executes INCREMENT register1      {register1 = 6}
S2: consumer executes LOAD register2, count    {register2 = 5}
S3: consumer executes DECREMENT register2      {register2 = 4}
S4: producer executes STORE count, register1   {count = 6}
S5: consumer executes STORE count, register2   {count = 4}
```

producer's view after S5 *(incorrect)*

consumer's view after S5 *(correct)*

HOUSTON, WE'VE GOT A PROBLEM !!!

Critical-Section Problem

- **Race Condition**
  - When there is concurrent access to shared data and the outcome depends upon the order of execution that is controlled by the OS and may vary over time.
- **Critical Section**
  - A section of code prone to race condition (where shared data is accessed).
- **Solution**:
  - Surround the critical section with:
    - **Entry Section**
      - Code that requests permission to enter its critical section.
      - “wait” (until safe to enter)
    - **Exit Section**
      - Code that is run after exiting the critical section
      - “signal” (done with it; somebody else can enter)

```
do {
    entry section: wait
    critical section
    exit section: signal
    remainder section

} while (true);
```

- We will look at a few approaches to solve this problem

Requirements for a Solution

- A solution to the critical section problem must satisfy **all the following conditions**:
  - **Mutual Exclusion**
    - If a process is executing its critical section, then no other processes can be executing theirs
  - **Progress**
    - If no process is executing its critical section and there exist some processes that wish to enter their critical sections, then the selection of the processes that will enter its critical section next cannot be postponed indefinitely
  - **Bounded Waiting**
    - After a process has made a request to enter its critical section, a bound must exist on the number of times that other processes can enter theirs before that request is granted

☛ Assume that each process executes at a nonzero speed
☛ No assumption concerning relative speed of the N processes

Solution 1: Peterson's Two-Process Solution

- The **two processes share** two variables:

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

- The variable **`turn`** indicates whose turn it is to enter the critical section.
- The **`flag`** array is used to indicate if a process is ready to enter the critical section.
  - **`flag[i] = true`** implies that process Process **`i`** is ready!

Process i

```
do {
   flag[i] = TRUE;
   turn = j;
   while (flag[j] && turn == j)
      ; // spin
   // critical section
   flag[i] = FALSE;
   // remainder section
} while (true);
```

wait

signal

- This solution assumes that computer does not reorder **`LOAD`** and **`STORE`** operations and that these instructions are **atomic**
  - that is, they cannot be interrupted

Peterson's Solution: Sample Run 1

time

Process 0

```
flag[0] = true;

turn = 1;
while (flag[1] && turn == 1) ;

while (flag[1] && turn == 1) ;

while (flag[1] && turn == 1) ;

// critical section

flag[0] = false;

flag[0] = true;
turn = 1;
while (flag[1] && turn == 1) ;
```

context switch

Process 1

```
flag[1] = true;
turn = 0;

while (flag[0] && turn == 0) ;

// critical section

flag[1] = false;

flag[1] = true;
turn = 0;
while (flag[0] && turn == 0) ;

while (flag[0] && turn == 0) ;

// critical section
```

Peterson's Solution: Sample Run 2

time

```
flag[0] = true;

turn = 1;

while (flag[1] && turn == 1) ;

// critical section

flag[0] = false;
```

```
flag[1] = true;

turn = 0;

while (flag[0] && turn == 0) ;

while (flag[0] && turn == 0) ;

while (flag[0] && turn == 0) ;

// critical section

flag[1] = false;
```

Implementing Bounded Waiting

- There is a problem with the previous solution, since a thread may starve waiting for its turn that is never granted
- In this solution, we order the access to the critical section, so every thread gets a chance
- In this way, we have a complete solution to the critical section problem

key to the critical section

request for access

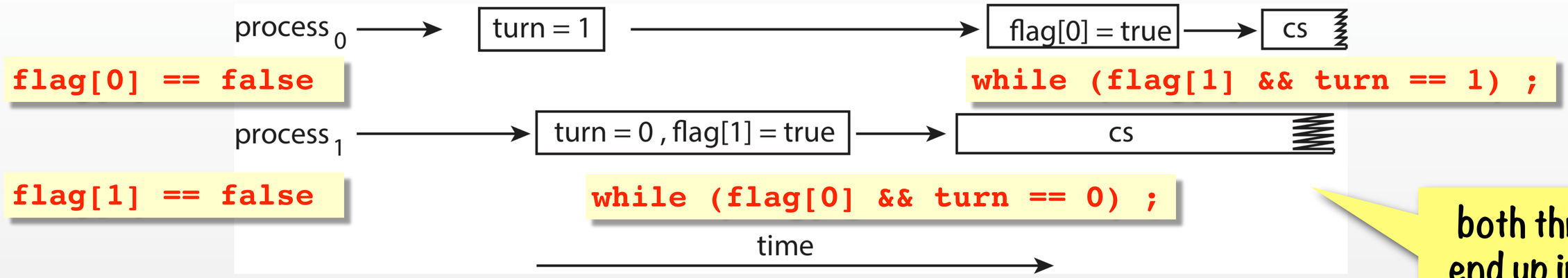
```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

   // 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);
```

evidently no process wants to get in, as we have made a full cycle looking for one

we found one whose request has not been satisfied yet

Issues with Peterson's Solution

- Modern multi-processor computers may reorder execution of **`LOAD`** and **`STORE`** operations for independent variables to speed up computation



both threads end up in their critical sections

time

```
turn = 1;

flag[0] = true;
while (flag[1] && turn == 1) ;

// critical section
```

```
turn = 0;
flag[1] = true;
while (flag[0] && turn == 0) ;

// critical section
```

General Synchronization Issues on MP Systems

Shared data

```
boolean flag = false;
int x = 0;
```

Process 0

```
while (!flag)
   ;
print x;
```

Process 1 Run 1

```
x = 100;
flag = true;
```

OR

Process 1 Run 2

```
flag = true;
x = 100;
```

- All is fine if there is no reordering as shown in Run 1 of Process 1: **`x`** gets set, then the **`flag`** is raised, so **`100`** can be printed as the value of **`x`** in Process 0
- However, if the system executes Process 1 as shown in Run 2, we have a problem: **`flag`** gets set before **`x`**, so Process 0 may print the old value of **`x`** rather than **`100`**
- Therefore, we need better tools for synchronization.
  - many systems provide hardware support

Memory Barriers (Fences)

- Memory models:
  - **strongly ordered**
    - no re-ordering of instructions
  - **weakly ordered**
    - re-ordering possible
  - depend on the processor type, so kernel cannot make assumptions
- **Memory barriers** (or **fences**) force ordering of **`LOAD`** and **`STORE`**
  - e.g.

```
$ man membarrier
```

Shared data

```
boolean flag = false;
int x = 0;
```

Process 1

```
while (!flag)
   membarrier();
print x;
```

Ensures that flag is loaded before x

Process 2 v.1

```
x = 100;
membarrier();
flag = true;
```

Ensures that x is stored before flag

Solution 2: Disabling Interrupts

- Context switches happen most commonly in response to interrupts

- Simple solution: disable interrupts when a process enters its critical section
  - currently running code would execute the critical section without preemption
    - as there would be no context switches

- Generally too inefficient
  - Uniprocessor
    - if we block all interrupts, then no event that needs attention is processed
      - e.g., no I/O operations
  - Multiprocessor
    - some processors may be idling rather than running code that is not in any critical section
    - I/O cannot be served even on on the idling processors

Solution 3: Locking Atomic Instructions

- Modern machines provide special atomic hardware instructions
  - recall: atomic == non-interruptible

```
UBUNTU:
$ sudo apt install libatomic-ops-dev
$ man libatomic-ops
```

- Either
  - atomically test memory word (“lock”) and set its value, or
  - atomically compare and swap contents of two memory words

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

- Implementation might be challenging on multiprocessor architectures

Test And Set (TAS) Atomic Instruction

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

```
// initially lock == false
do {
   while (test_and_set(&lock))
      ; // spin

   // critical section

   lock = false;

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

Compare And Swap (CAS) Atomic Instruction

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

```
// initially lock == false
do {
   while (compare_and_swap(&lock, false, true))
      ; // spin

   // critical section

   lock = false;

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

The Busy-Waiting Problem

- The solutions to the critical section problem that we saw so far have problem in that they loop continuously in the entry code
  - also called **Spin-Lock Problem** (process is "spinning" while waiting)
    - locks that work in this way are also called **spinlocks**

```
while (flag[j] && turn == j)
       ; // spin
// critical section
```

- Waste of CPU cycles, so process could be blocked instead until allowed to enter
  - however, blocking is expensive, so in spite of their shortcomings spinlocks are popular and commonly used (e.g., Linux):
    - no blocking, so no expensive overhead
    - there is still "normal" scheduling of the waiting process, but it may be relatively more efficient
    - might be acceptable if threads lock critical sections for very short periods
  - can also be acceptable in multi-processor applications where some processors can spin-lock while others continue to execute

Solution 4: Semaphores

- **Semaphore** S – integer variable with two indivisible (atomic) standard operations to modify it:
  - **wait()** and **signal()**
    - originally called **P()** and **V()** **(after Dijkstra; a Dutch)**
    - **acquire()** and **release()** in Java
  - cannot be accessed in any other way
- Less complicated than earlier methods

Proberen - test

Verhogen - increase

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

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

```
Semaphore count;

// count initialized to 1
do {
   wait (count);

   // critical section

   signal (count);

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

- **Counting semaphore**
  - integer value can range over an unrestricted domain

Binary Semaphore

- **Binary semaphore** a.k.a. **mutex** (**mut**ual **ex**clusion)
  - the value can only be 0 or 1

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

```
signal (S)
{
  S = 1;
}
```

```
Mutex mutex;

// mutex initialized to 1
do {
  wait (mutex);

  // critical section

  signal (mutex);

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

We still have a busy-waiting problem however!

Pthread Semaphores

```
Semaphore sem;
do {
   wait (sem);

   // Critical Section

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

recall

Explore: `semaphore.c`

```
sem_t sem;
pthread_t *threads;

// shared resource
int r;

void *bee(void *param)
{
   ...
   while (numberOfIterations-- > 0)
   {
       // wait
       sem_wait(&sem);
       // in critical section

       r = rand() % sleepCeiling;
       printf("This bee is bziing for %d seconds :-)\n", r);
       ...
       sleep(r);
       ...
       // signal
       sem_post(&sem);
       // out of critical section
   }
}
```

```
int main(int argc, char *argv[])
{
   ...

   // initialize the semaphore
    if (sem_init(&sem, 0, 0) < 0)
       oops("Semaphore initialization");

    // Create a bunch of bees
   threads = (pthread_t *) calloc(numberOfBees, sizeof(pthread_t));

   for(i = 0; i < numberOfBees; i++)
   {
       if (pthread_create(threads + i, NULL, bee, NULL) < 0)
          oops("Semaphore initialization");
   }

    // signal
    sem_post(&sem);

    // wait for the bees
   for(i = 0; i < numberOfBees; i++)
   {
       if (pthread_join(threads[i], NULL) < 0)
          oops("Semaphore initialization");
    }

   // destroy the semaphore
    if (sem_destroy(&sem) < 0)
       oops("Semaphore initialization");

   ...
}
```

needed as the "first push"

Semaphores without Busy-Waiting

- Associate a waiting queue with each semaphore, so it now has
  - value (of type integer; like before)
  - a pointer to the waiting queue
- Two blocking operations:
  - **block**
    - place the process invoking the operation on the appropriate waiting queue, and then block
  - **wakeup**
    - remove one of processes in the waiting queue and place it in the ready queue (wake it up)
      - recall scheduling algorithms!

- Implementation of **`wait()`**:

```
wait(semaphore *S)
{
  S->value--;
  if (S->value < 0) {
    add this process to S->list;
    block();
  }
}
```

abs(S->value) indicates the number of blocked processes in the queue

- Implementation of **`signal()`**:

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

can also be used with mutex

Hybrid Approaches to Busy-Waiting Problem

- A need to strike a balance between the cost of blocking a waiting process and scheduling as it is allowed to spin
- Possible approaches:
  - OS may use spinlock approach first, but if used for long, it may force blocking
  - OS may spin if the thread holding the lock is running, but switch to blocking when the thread holding the lock is being blocked
    - i.e., when no immediate chance for CPU release

- For example, adaptive mutex in pthread library:

```
// pthread's adaptive mutex
. . .
pthread_mutexattr_t mutex_attr;
pthread_mutexattr_init(&mutex_attr);
pthread_mutexattr_settype(&mutex_attr, PTHREAD_MUTEX_ADAPTIVE_NP);
pthread_mutex_init(&mutex, &mutex_attr);
. . .
```

A Note on Semaphore Implementation

- Must guarantee that no two processes can execute wait() and signal() on the same semaphore at the same time
- Thus, **implementation of these methods becomes itself the critical section problem** where the wait() and signal() code is placed in a critical section
  - that may result in busy waiting in critical section implementation
  - however, execution of the code of wait() and signal() is fast usually, so there is little busy-waiting
    - in contrast to applications that may spend lots of time in critical sections
      - therefore, spinning/busy waiting is not a good programming practice in general.

```
wait (S) {
   wait(M);
   S--; // critical section
   signal(M);
   while (S < 0)
      ; // no-op
}

signal (S) {
  wait(M);
  S++; // critical section
  signal(M);
}
```

Potential Problem with Semaphores

- Semaphores with blocking provide an improvement, but… they are programmer's tools, and programmers make mistakes
  - For example, instead of the correct:

```
wait(mutex)
…
signal(mutex)
```

  - a programmer may use the following legal but semantically incorrect sequences of semaphore operations:

```
signal(mutex)
…
wait(mutex)
```

```
wait(mutex)
…
wait(mutex)
```

  - or she/he may omit one or both calls altogether

- That would lead to problems that are usually not easily tracked
  - e.g., multithreaded programs may seem to work fine although some functionality may be broken by stalled threads

Deadlock and Starvation

- **Deadlock**
  - two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
  - Example:
    - Let S and Q be two semaphores initialized to 1, and P0 and P1 two processes using them:

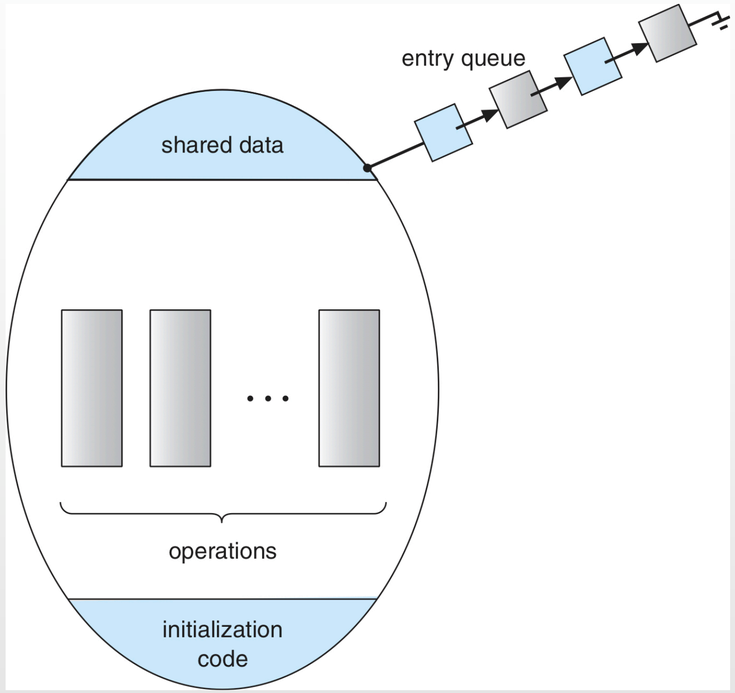
```
        P0                  P1
    wait(S);            wait(Q);
    wait(Q);            wait(S);
    .                       .
    .                       .
    .                       .
    signal(S);          signal(Q);
    signal(Q);          signal(S);
```

- **Starvation**
  - indefinite blocking
  - a process may never be removed from the semaphore queue in which it is suspended.

Solution 5: Monitors

- A high-level abstraction that provides a convenient and effective **high-level language** mechanism for process synchronization
- Only one process may be active within the monitor at a time

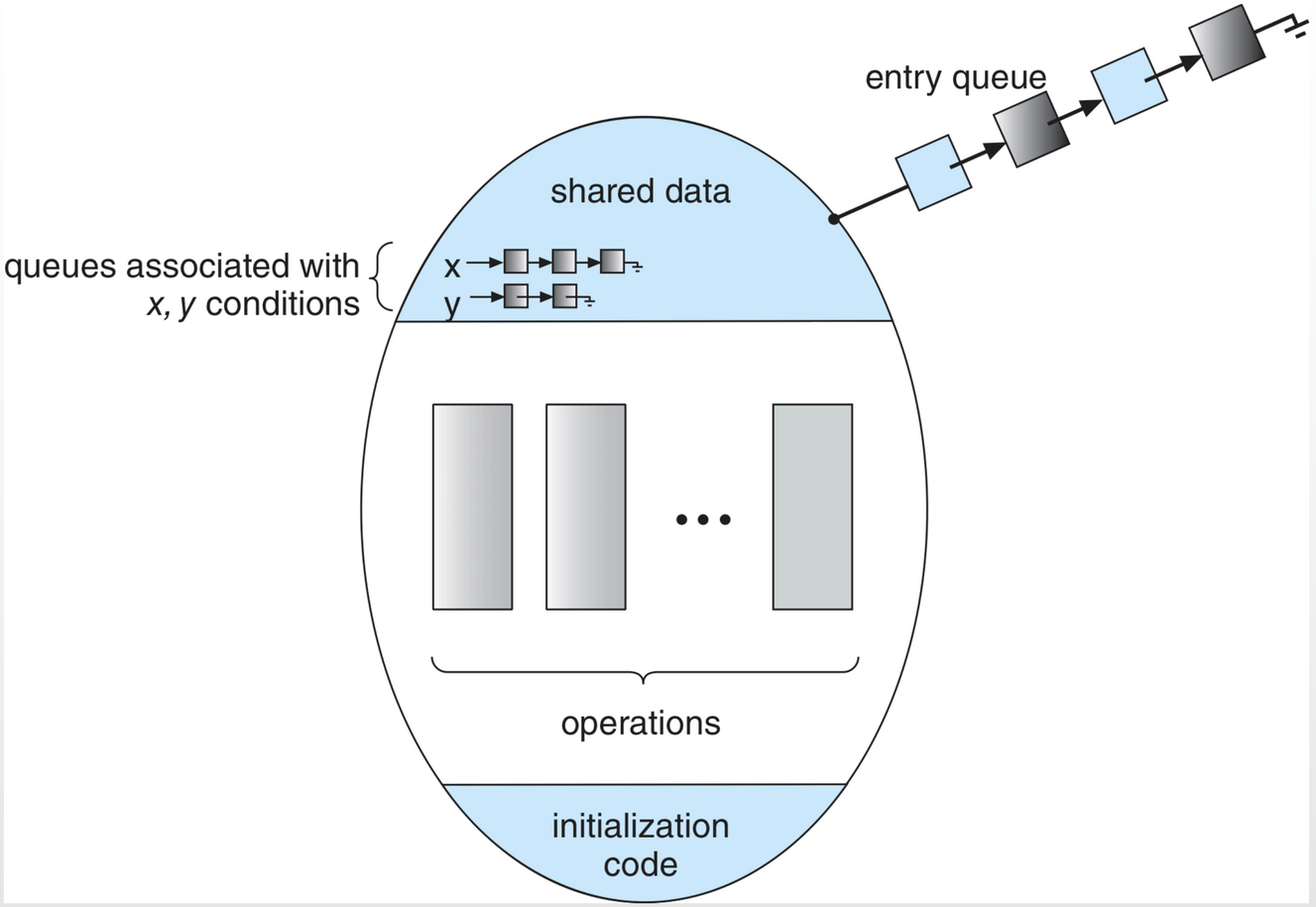
```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (...) {...}
          ...

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

  Initialization code (...) {...}
          ...
  }
}
```



- **pthreads library does not have monitors**
  - but they can be simulated using semaphores - as you will do in the lab!

Monitor with Condition Variables

- Variables associated with locks (mutexes) allow for finer selectivity through two operations:
  - **`cond_wait()`** – a thread that invokes the operation is suspended
  - **`cond_signal()`** – resumes one of the suspended threads (those that executed `cond_wait()`; if any)



Pthread Condition Variables

- pthread library supports condition variables

```
void *watcher(void *arg) {
...
   pthread_mutex_lock(&countMutex);
...
   pthread_cond_wait(&countTargetConditionVariable, &countMutex);
...
   pthread_mutex_unlock(&countMutex);
}
void *counter(void *arg) {
...
   pthread_mutex_lock(&countMutex);
...
   if (++count > countingTarget)
      pthread_cond_signal(&countTargetConditionVariable);
...
   pthread_mutex_unlock(&countMutex);
}
int main() {
   pthread_mutex_init(&countMutex, NULL);
   pthread_cond_init (&countTargetConditionVariable, NULL);
...
   for (int i = 0; i < numberOfCounters; i++)
      pthread_create(threads + i, ..., counter, ...);

   pthread_create(threads + i, ..., watcher, ...)
...
   pthread_mutex_destroy(&countMutex);
   pthread_cond_destroy(&countTargetConditionVariable);
}
```

Explore: `condvar.c`

must be associated with a semaphore, and can be used only inside the critical section guarded by that semaphore

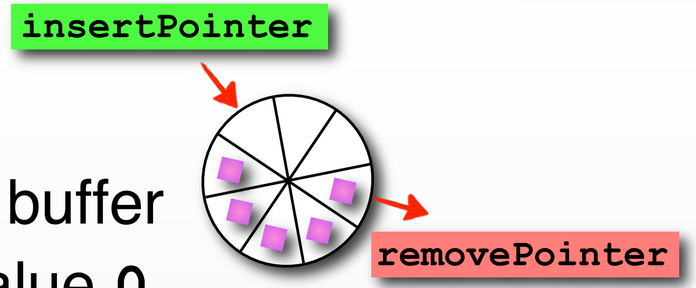
Classical Synchronization Paradigms

- We will look at a number of classical problems (**paradigms**) that illustrate the ideas for process synchronization

- Bounded-Buffer Paradigm
- Readers and Writers Paradigm
- Dining-Philosophers Paradigm

Bounded-Buffer Problem

- **N** buffers, each can hold one item
  - a.k.a. circular buffer problem: one buffer with **N** slots
- Binary semaphore **mutex** initialized to the value **1** - protects the buffer
- Counting semaphore **loaded** (“loaded slots“) initialized to the value **0**
- Counting semaphore **empty** (“empty slots”) initialized to the value **N**



In general, any process can be a producer or a consumer at any given time

producer’s index to the buffer

buffer.c
buffer.h

```
do {
   // produce an item in insertPointer

   wait (empty); // for at least one
   wait (mutex);

   //  add the item to the buffer

   signal (mutex);
   signal (loaded);

} while (true);
```

Producer

```
do {
   wait (loaded); // for at least one loaded
   wait (mutex);

   // remove an item from  buffer to removePointer

   signal (mutex);
   signal (empty);

   // consume the item in removePointer

} while (true);
```

consumer’s index to the buffer

Consumer

Readers-Writers Paradigm

- 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.

- Similar to producer/consumer, but:
  - we allow multiple readers to read at the same time as they do not change the values
  - only one single writer can access the shared data at the same time, since writers are the ones that actually modify the data
  - no reading can occur when writing is going on

Reader-Writers Solution

- Shared Data
  - data set
  - semaphore **`mutex`** initialized to 1
  - semaphore **`wrt`** initialized to 1
  - integer **`readerCount`** initialized to 0

writer

```
do {
   wait (wrt) ;

   // writing is performed

   signal (wrt) ;

} while (TRUE);
```

For a solution with pthread's read-write locks explore:
`readwrite.c`

reader

```
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);
```

critical section for readcount

first reader ensures that no writing is in progress while reading

writing is enabled when there are no committed readers

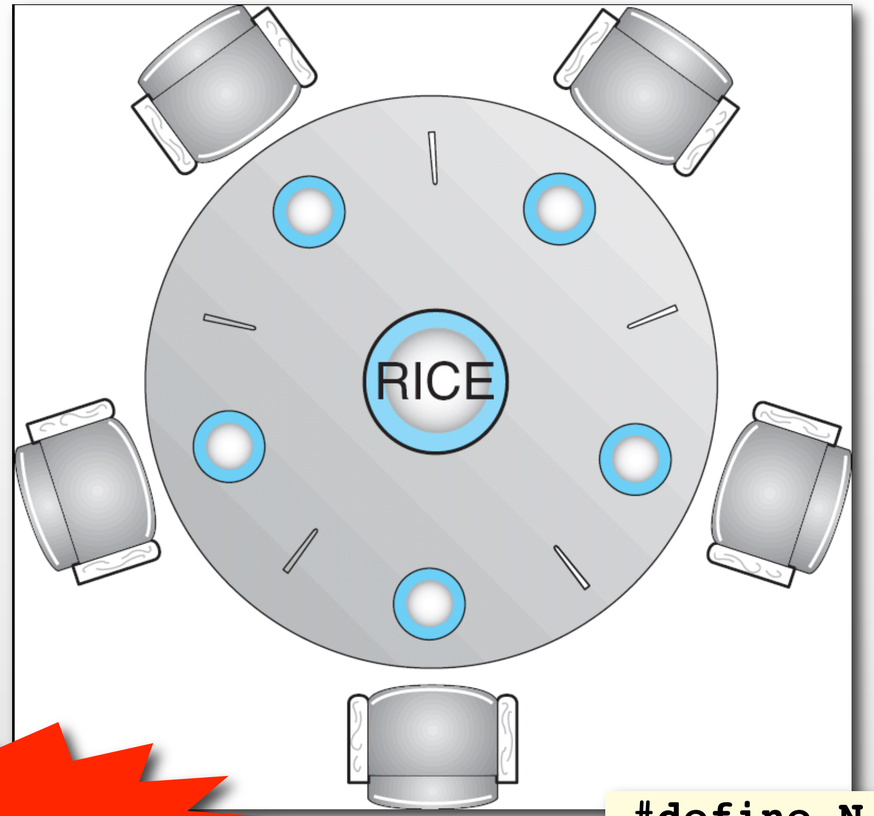
critical section for readcount

Dining Philosophers (DP) Problem

- Shared data
  - A bowl of rice (data set)
  - Semaphore **`chopstick[N]`** initialized to **`1`**

- The structure of philosopher **`i`**:

```
do  {
   wait ( chopstick[i] );
   wait ( chopstick[ (i + 1) % N] );

   //  eat

   signal ( chopstick[i] );
   signal ( chopstick[ (i + 1) % N] );

   //  think

} while (TRUE);
```




```
#define N 5
```

Danger of a deadlock!
Why?

Monitor-based Solution Of the DP Problem

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

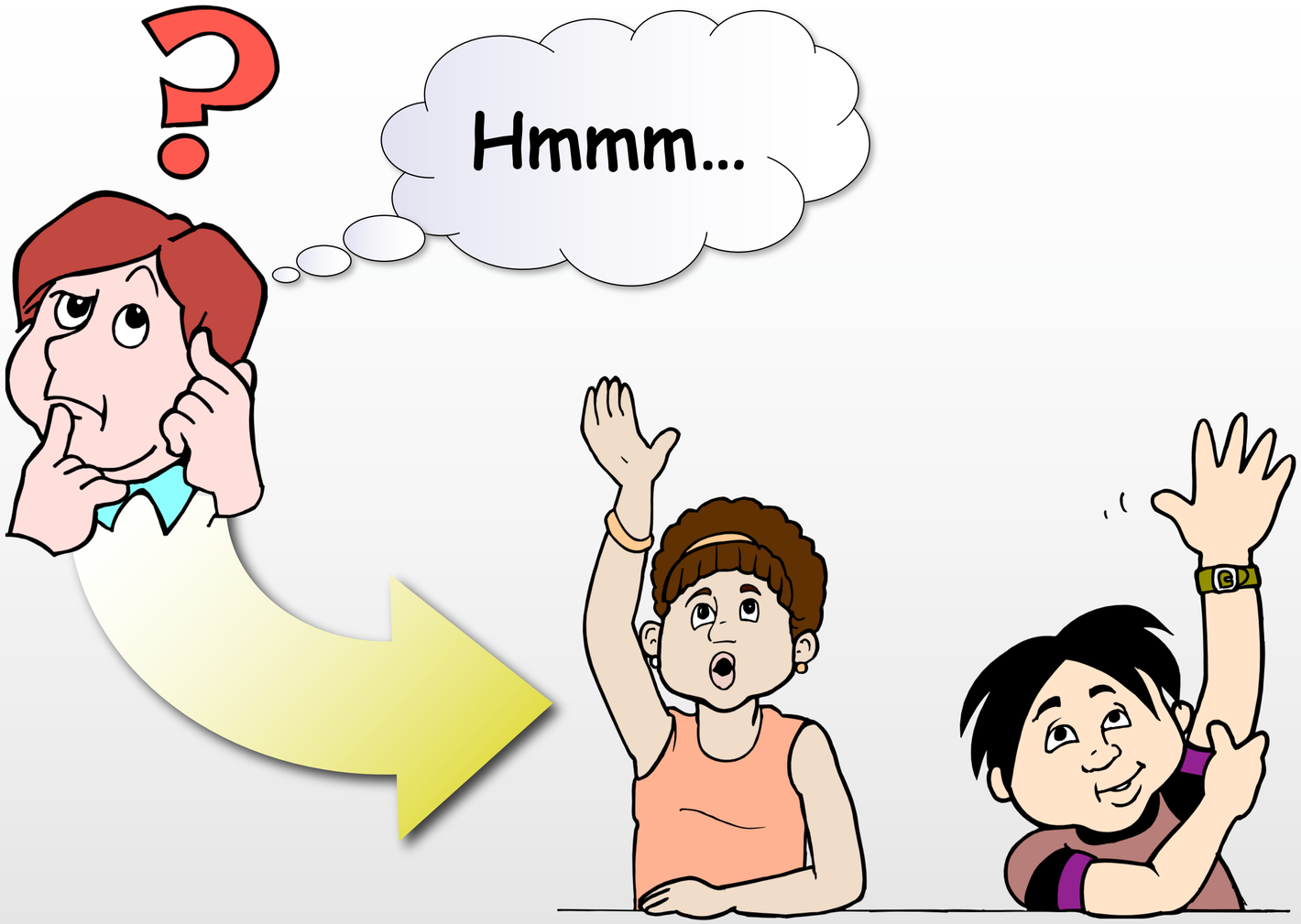
```
DiningPhilosophers.pickup (i);
EAT
DiningPhilosophers.putdown (i);
```

- Remember:
  the monitor guarantees exclusive execution of the methods!

```
monitor DiningPhilosophers {
   enum { THINKING; HUNGRY; EATING} state[N];
   condition self[N];

   void pickup (int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING) self[i].wait();
   }
   void putdown (int i) {
      state[i] = THINKING;
      // test left and right neighbors
      test((i + (N-1)) % N);
      test((i + 1) % N);
   }
   void test (int i) {
      if ( (state[(i + (N-1)) % N] != EATING) &&
           (state[i] == HUNGRY) &&
            (state[(i + 1) % N] != EATING) ) {
         state[i] = EATING ;
         self[i].signal();
      }
   }
   initialization_code() {
      for (int i = 0; i < N; i++)
         state[i] = THINKING;
   }
}
```

note that `test(i)` can be also called by the neighbors of `i` (see `putdown()`)

this might be done by a neighbor that has just finished eating and is executing `putdown()`



COMP362 Operating Systems

Prof. AJ Bieszczad