Interprocess Communication

Today

- Race condition & critical regions
- Mutual exclusion with busy waiting
- Sleep and wakeup
- Semaphores and monitors
- Classical IPC problems

Next time

- Deadlocks

Cooperating processes

- Cooperating processes need to communicate
  - Coop processes – can affect/be affected by others
- Issues

  1. How to pass information to another process?
  2. How to avoid getting in each other's ways?
     - Two processes trying to get the last seat on a plane
  3. How to ensure proper sequencing when there are dependencies?
     - Process *A* produces data, while *B* prints it – *B* must wait for *A* before starting to print
- How about threads?

  1. Easy

  2 & 3. Pretty much the same

Race conditions

- Many times cooperating process share memory
- A common example – print spooler
  – A process wants to print a file, enter file name in a special spooler directory
  – Printer daemon, another process, periodically checks the directory, prints whatever file is there and removes the name

A: $next\_slot_A \leftarrow in$ % 7

Switch

B: $next\_slot_B \leftarrow in$ % 7

B: $spooler\_dir[next\_slot_B] \leftarrow file\_name_B$

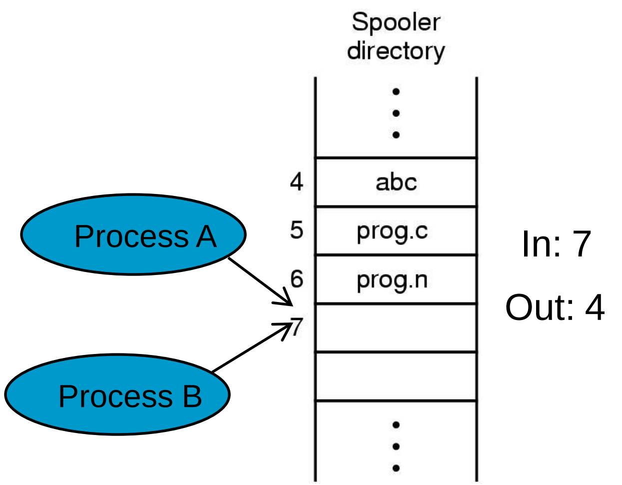
Switch

B: $in \leftarrow next\_slot_B + 1$ % 8

Switch

A: $spooler\_dir[next\_slot_A] \leftarrow file\_name_A$

A: $in \leftarrow next\_slot_A + 1$ % 8



- Race Condition:
  – Two or more processes access (r/w) shared data
  – Final results depends on order of execution

Critical regions & mutual exclusion

- Problem – race condition
- Where in code? Critical region (CR)
- We need a way to ensure that *if a process is using a shared item (e.g. a variable), other processes will be excluded from doing it*

  Mutual exclusion
- 1. No two processes simultaneously in CR
- But there's more - a good solution must also ensure ...

  2. No assumptions on speeds or numbers of CPUs

  3. No process outside its CR can block another one

  4. No process should wait forever to enter its CR

Ensuring mutual exclusion

- Lock variable?
  - Lock initially 0
  - Process checks lock when entering CR
  - *Problem?*
- Disabling interrupts
  - Simplest solution
  - *Problems?*
    - Users in control
    - Multiprocessors?
  - Use in the kernel

Strict alternation

- Taking turns
  - `turn` keeps track of whose turn it is to enter the CR

Process 0

```
while(TRUE) {
  while(turn != 0);
  critical_region0();
  turn = 1;
  noncritical_region0();
}
```

Process 1

```
while(TRUE) {
  while(turn != 1);
  critical_region1();
  turn = 0;
  noncritical_region1();
}
```

- Problems?
  - What if process 0 sets turn to 1, but it gets around to just before its critical region before process 1 even tries?
  - Violates conditions 3

Peterson’s solution

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

int turn;
int interested[N];

void enter_region(int process)
{
  int other;

  other = 1 - process;
  interested[process] = TRUE;
  turn = process;
  while (turn == process &&
         interested[other] == TRUE);
}

void leave_region(int process)
{
  interested[process] = FALSE;
}
```

Template of a process’ access to the critical region (process 0):

```
…
enter_region(0);
<CR>
leave_region(0);
…
```

Tracing Peterson's

Process 0	Common variables	Process 1
enter_region(0) other = 1 interested[0] = T turn = 0 **(Process 0 in)**	interested[0] = F interested[1] = F, turn = ?	
	interested[0] = T, interested[1] = F, turn = 0	--- --- --- --- ---
--- ---		
		--- --- ---

```
void enter_region(int process)
{
  int other;
  other = 1 – process;
  interested[process] = TRUE;
  turn = process;
  while (turn == process &&
       interested[other] == TRUE);
}
```

Tracing Peterson's

Process 0	Common variables	Process 1
enter_region(0) other = 1	interested[0] = F interested[1] = F, turn = ?	
	interested[0] = F interested[1] = T, turn = ?	enter_region(1) other = 0 interested[1] = T
interested[0] = T turn = 0	interested[0] = T interested[1] = T, turn = 0	
	interested[0] = T Interested[1] = T, turn = 1	turn = 1 <Busy Wait>
turn != 0 *<CR>* leave_region(0) interested[0] = F	interested[0] = F, interested[1] = T, turn = 1	
	interested[0] = F, Interested[1] = F, turn = 1	<CR>

```
void enter_region(int process)
{
  int other;
  other = 1 – process;
  interested[process] = TRUE;
  turn = process;
  while (turn == process &&
        interested[other] == TRUE);
}
```

TSL(test&set) -based solution

- With a little help from hardware – TSL instruction
- Atomically test & modify the content of a word

```
TSL REG, LOCK
```

  - REG ← LOCK >> Read the content of variable LOCK into register REG
  - LOCK ← non-zero value >> Set lock to a non-zero value

- Entering and leaving CR

```
enter_region:
   TSL REGISTER, LOCK
   CMP REGISTER, #0
   JNE enter_region | non zero, lock set
   RET | return to caller, you're in
```

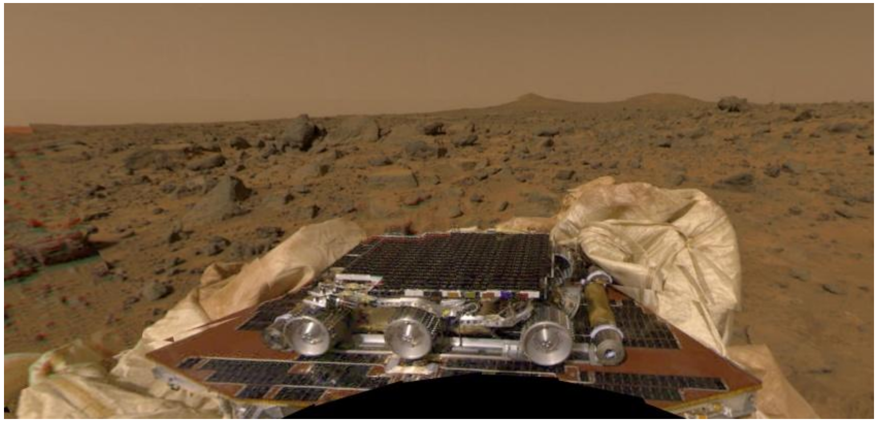
Busy waiting

```
leave_region:
   MOVE LOCK, #0
   RET
```

- A lock that uses busy waiting – *spin lock*

Busy waiting and priority inversion

- Problems with Peterson and TSL-based approach?
  - Waste CPU by busy waiting
  - Can lead to *priority inversion*
    - Two processes, H (high-priority) & L (low-priority)
    - L gets into its CR
    - H is ready to run and starts busy waiting
    - L is never scheduled while H is running …
    - *So L never leaves its critical region and H loops forever!*
- *Welcome to Mars!*
  - Mars Pathfinder
    - Launched Dec. 4, 1996
    - Landed July 4th, 1997



Problems in the Mars Pathfinder*

- Periodically the system reset itself, loosing data
- VxWork provides preemptive priority scheduling
- Pathfinder software architecture
  - An information bus with access controlled by a lock
  - A bus management (B) high-priority thread
  - A meteorological (M) low-priority, short-running thread
    - If B thread was scheduled while the M thread was holding the lock, the B thread busy waited on the lock
  - A communication (C) thread running with medium priority
- **Sometimes, C was scheduled while B was waiting on M**
- After a bit of waiting, a watchdog timer would reset the system ☺
- How would you fix it?
  - Priority inheritance – the M thread inherits the priority of the B thread blocked on it
  - Actually supported by VxWork but dissabled!

*As explained by D. Wilner, CTO of Wind River Systems, and narrated by Mike Jones

Sleep & wakeup

- An alternative solution
  - Sleep – causes the caller to block
  - Wakeup – process pass as parameter is awakened
- Producer-consumer (aka bounded buffer) example
  - Two processes & one shared, fixed-size buffer

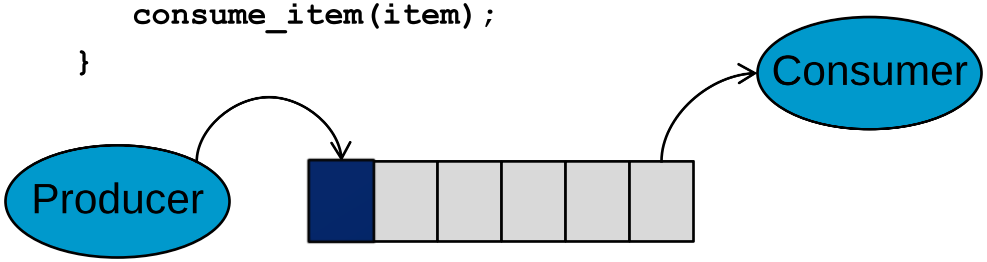
Producer

```
while (TRUE){
   item = produce_item();
   while (count == N);
   insert_item(item);
   ++count;
   if (count == 1)
      wakeup(consumer)
}
```

Consumer

```
while (TRUE){
   while(count == 0);
   item = remove_item();
   --count;
   if (count == (N -1))
      wakeup(producer);
   consume_item(item);
}
```

Consumer is not yet logically sleep - producer's signal is lost!



Semaphores

- A variable atomically manipulated by two operations – down (P) & up (V)
- Each semaphore has an associated queue of processes/threads
  - P/wait/down(sem)
    - If sem was “available” (>0), decrement sem & let thread continue
    - If sem was “unavailable” (<=0), place thread on associated queue; run some other thread
  - V/signal/up(sem)
    - If thread(s) are waiting on the associated queue, unblock one (place it on the ready queue)
    - If no threads are waiting, increment sem
      - The signal is “remembered” for next time up(sem) is called
    - Might as well let the “up-ing” thread continue execution
- Semaphores thus have history

Semaphores

- Abstract implementation

down(S):

```
    --S.value;
    if (S.value < 0){
        add this process to S.L;
        block;
    }
```

up(S):

```
    S.value++;
    if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
    }
```

```
typedef struct {
   int value;
   struct process *L;
} semaphore;
```

- With multiple CPUs – lock semaphore with TSL
- *But then how's this different from previous busy-waiting?*

Semaphores

Operation	**Value**	**S.L.**
P1 down	0	{}
P2 down	-1	{P2}
P3 down	-2	{P2,P3}
P1 up	-1	{P3}

```
down(S):
--S.value;
if (S.value < 0){
   add this process to S.L;
   block;
}

up(S):
S.value++;
if (S.value <= 0) {
    remove a process P from S.L;
    wakeup(P);
}
```

Semaphores

`empty = # available slots, full = 0, mutex = 1`

Producer

```
while (TRUE){
    item = produce_item();
    down(empty);
    down(mutex);
    insert_item(item);
    up(mutex);
    up(full);
}
```

Consumer

```
while (TRUE){
    down(full);
    down(mutex);
    item = remove_item();
    up(mutex);
    up(empty);
    consume_item(item);
}
```

- Semaphores and I/O devices

Mutexes

- Two different uses of semaphores
  - Synchronization – full & empty
  - Mutex – used for mutual exclusion

- Useful w/ thread packages
- Other possible operation

```
mutex_trylock()
```

```
mutex_lock:
    TSL REGISTER, MUTEX
    CMP REGISTER, #0
    JXE ok
    CALL thread_yield
    JMP mutex_lock
ok: RET

mutex_unlock:
   MOVE MUTEX, #0
   RET
```

Problems with semaphores

- Can be used to solve all of the traditional synchronization problems, but:
  - Semaphores are essentially shared global variables
    - Can be accessed from anywhere (bad software engineering)
  - No connection bet/ the semaphore & the data controlled by it
  - Used for both critical sections & for coordination (scheduling)
  - No control over their use, no guarantee of proper usage

Producer

```
while (TRUE){
    item = produc
    down(mutex);
    down(empty);
    insert_item(item);
    up(mutex);
    up(full);
}
```

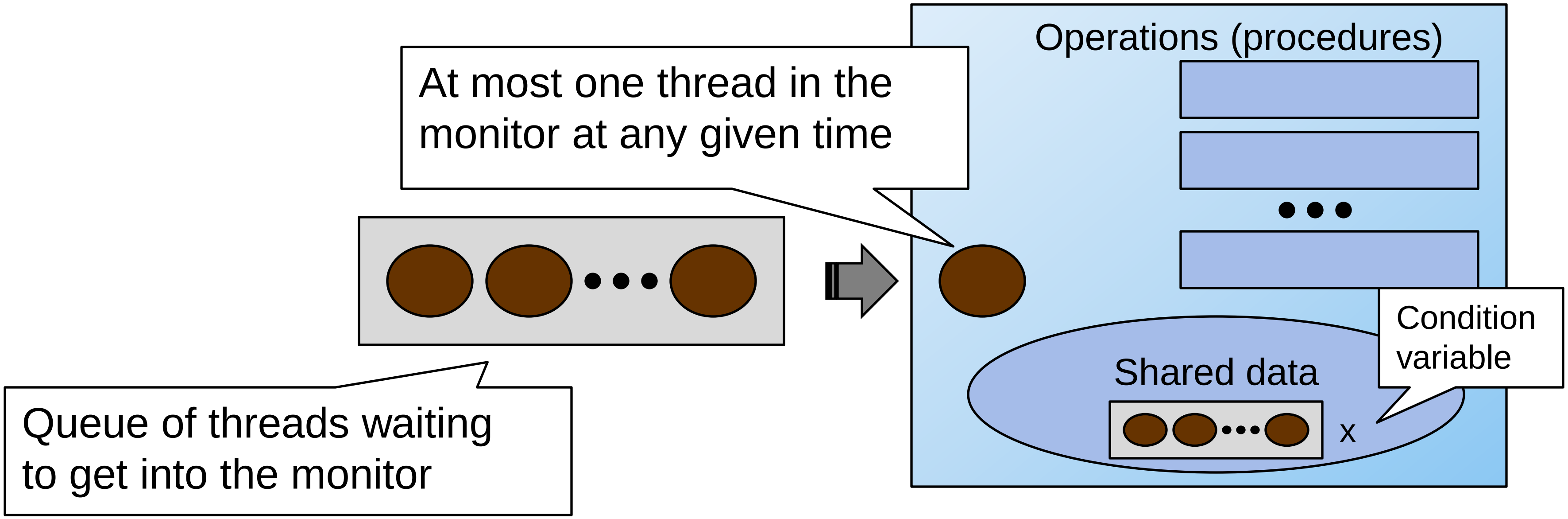
What happens if the buffer is full?

Consumer

```
while (TRUE){
    down(full);
    down(mutex);
    item = remove_item();
    up(mutex);
    up(empty);
    consume_item(item);
}
```

Monitors

- Monitors - higher level synchronization primitive
  - A programming language construct
    - Collection of procedures, variables and data structures
  - Monitor's internal data structures are private
- Monitors and mutual exclusion
  - Only one process active at a time - *how?*
  - Synchronization code is added by the compiler



Monitors

- Once inside a monitor, a process/thread may discover it can't continue, and want to wait, or inform another one that some condition has been satisfied
- To enforce sequences of events? Condition variables
  - Two operations – `wait & signal`
  - Condition variables can only be accessed from within the monitor
  - A thread that waits "steps outside" the monitor (to a wait queue associated with that condition variable)
  - What happen after the signal?
    - Hoare – process awakened run, the other one is suspended
    - Brinch Hansen – process doing the signal must exit the monitor
    - *Third option? Mesa programming language*
  - `Wait` is not a counter – signal may get lost

Producer-consumer with monitors

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count +
        if count = 1 the
    end;
    function remove: inte
    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

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

Why is OK here and not with sleep/wakeup?

The consumer could never run before the wait completes!

Producer-consumer with message passing

- IPC in distributed systems
- Message passing

  `send(dest, &msg)`

  `recv(src, &msg)`
- Design issues
  - Lost messages: acks
  - Duplicates: sequence #s
  - Naming processes
  - Performance
  - …

```
#define N 100    /* num. of slots in buffer */

void producer(void)
{
   int item; message m;

   while(TRUE) {
      item = produce_item();
      receive(consumer, &m);
      build_message(&m, item);
      send(consumer, &m);
    }
}

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

   for(i = 0; i < N; i++) send(producer, &m);

   while(TRUE) {
       receive(producer, &m);
       item = extract_item(&m);
       send(producer, &m);
       consume_item(item);
     }
}
```

Readers-writers problem

- Model access to database
- One shared database
- Multiple readers allowed at once
- If writers is in, nobody else is

```
void reader(void)
{
    while(TRUE) {
        down(&mutex);
        ++rc;
        if (rc == 1) down(&db);
        up(&mutex);

        read_db();

        down(&mutex);
        --rc;
        if (rc == 0) up(&db);
        up(&mutex);

        use_data();
    }
}
```

```
void writer(void)
{
    while(TRUE) {
      think_up_data();
      down(&db);
      write_db();
      up(&db);
    }
}
```

What problem do you see for the writer?

Idea for a solution: When a reader arrives, if there's a writer waiting, the reader could be suspended behind the writer instead of being immediately admitted.

Dining philosophers problem

- Philosophers eat/think
- To eat, a philosopher needs 2 chopsticks
- Picks one at a time
- *How to prevent deadlock*

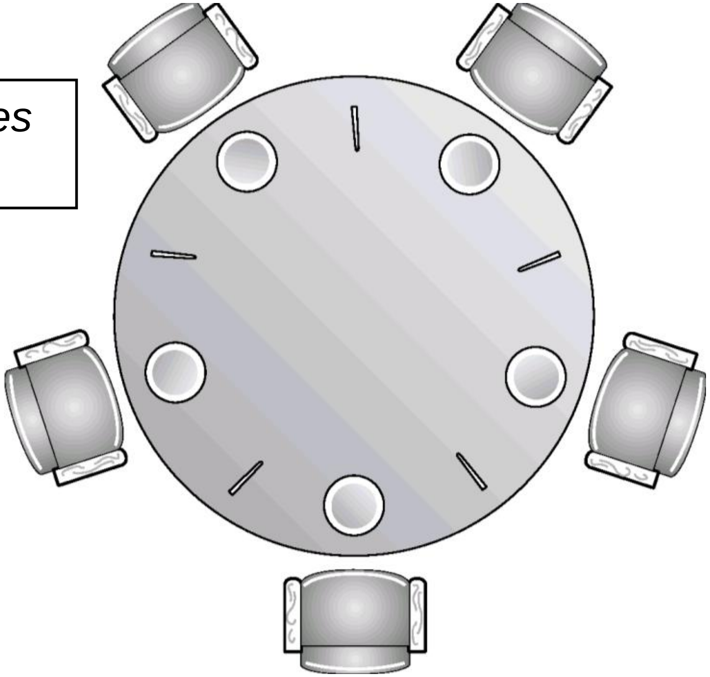
```
#define N 5

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_chopstick(i);
        take_chopstick((i+1)%N);
        eat();
        put_chopstick(i);
        put_chopstick((i+1)%N);
    }
}
```

Now: Everybody takes the left chopstick!

Why not just protect all this with a mutex?

Nonsolution



Dining philosophers example

```
void philosopher(int i)
{
  while(TRUE) {
    think();
    take_chopstick(i);
    eat();
    put_chopstick(i);
  }
}

void take_chopstick(int i)
{
   down(&mutex);
   state[i] = HUNGRY;
   test(i);
   up(&mutex);
   down(&s[i]);
}
```

```
void put_chopstick(int i)
{
  down(&mutex);
  state[i] = THINKING;
  test(LEFT);
  test(RIGHT);
  up(&mutex);
}

void test(int i)
{
  if ((state[i] == hungry &&
     state[LEFT] != eating &&
     state[RIGHT] != eating) {
     state[i] = EATING;
     up(&s[i]);
  }
}
```

state[] – too keep track of philosopher's state (eating, thinking, hungry)

s[] – array of semaphores, one per philosopher

Dining philosophers with monitors

```
void philosopher(int i)
{
  while(TRUE) {
    dp.take_chopstick(i);
    eat();
    dp.put_chopstick(i);
  }
}

Monitor dp
{
  enum {EATING, HUNGRY, EATING}
   state[5];
  condition s[5];

  void take_chopstick(int i)
  {
     state[i] = HUNGRY;
     test(i);
     if (state[i] != EATING)
       s[i].wait();
  }
```

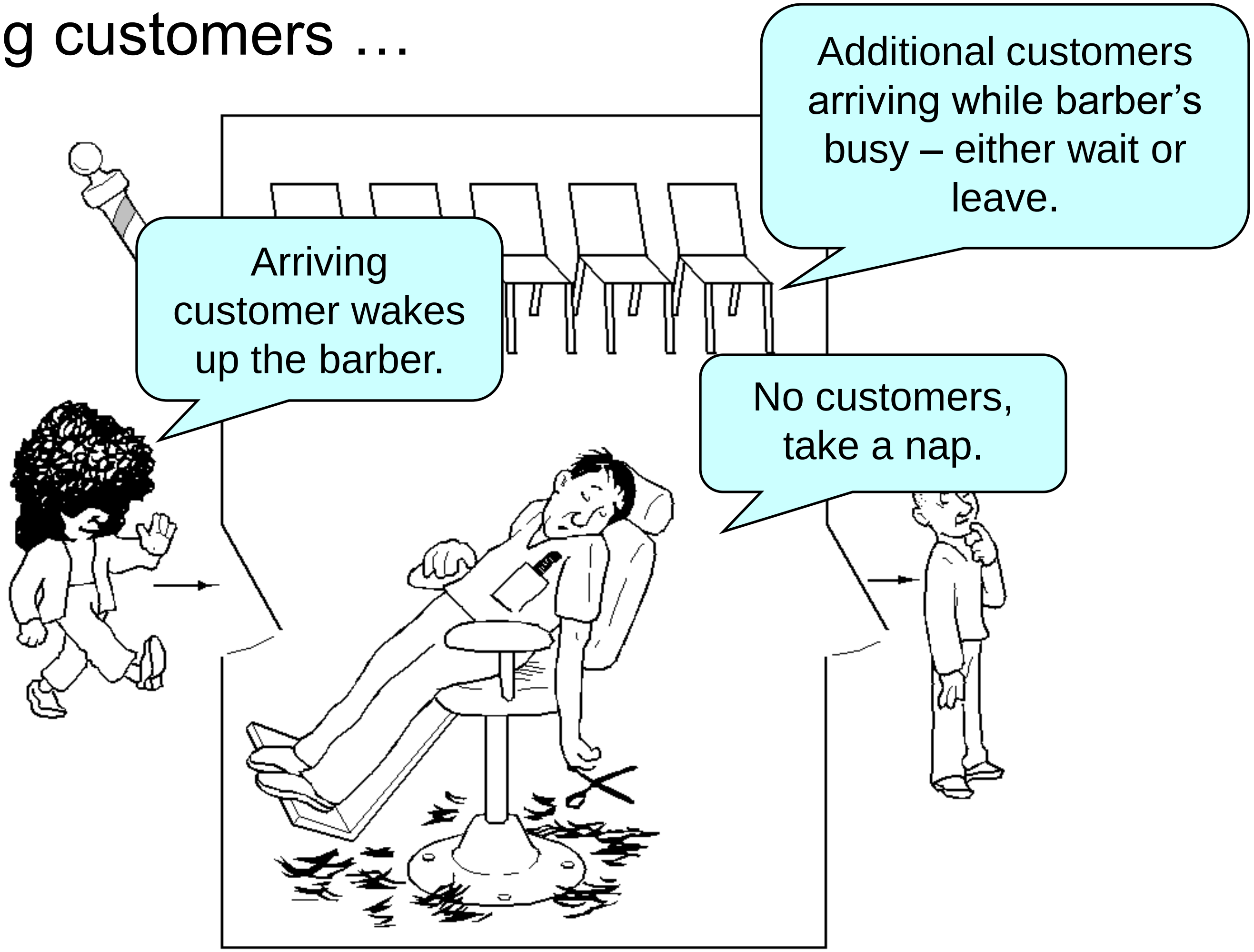
```
  void put_chopstick(int i)
  {
    state[i] = THINKING;
    test(LEFT); test(RIGHT);
  }

  void test(int i)
  {
    if ((state[i] == HUNGRY &&
         state[LEFT] != EATING &&
         state[RIGHT] != EATING) {
         state[i] = EATING;
         s[i].signal();
    }
  }

  void setup()
  {
    for (i = 0; i < 5; i++)
      state[i] = THINKING;
  }
} /* end Monitor dp */
```

The sleeping barber problem

One barber, one barber chair and *n* chairs for waiting customers …



The sleeping barber problem

```
#define CHAIRS 5

void barber (void)
{
   while (TRUE) {
     …
     …
     /* sleep if no customers */
     --waiting;
     …
     …
     cut_hair();
   }
}
```

```
void customer (void)
{
   …
   if (waiting < CHAIRS) {
       ++waiting; /* sit down */
       …
       …
       …
       get_haircut();
   } else { /* go elsewhere */
       …
   }
}
```

Semaphores:

- Customer – count waiting customers (excluding the one in the barber chair)
- Barbers – number of barbers who are idle
- mutex – for mutual exclusion

The sleeping barber problem

```
#define CHAIRS 5

void barber (void)
{
   while (TRUE) {
     down(&customers);
     /* sleep if no customers */
     down(&mutex);
     --waiting;
     up(&barbers);
     up(&mutex);
     cut_hair();
   }
}
```

```
void customer (void)
{
   …
   if (waiting < CHAIRS) {
       ++waiting; /* sit down */
       …
       …
       …
       get_haircut();
   } else { /* go elsewhere */
       …
   }
}
```

Semaphores:

- Customer - count waiting customers (excluding the one in the barber chair)
- Barbers – number of barbers who are idle
- mutex – for mutual exclusion

The sleeping barber problem

```
#define CHAIRS 5

void barber (void)
{
   while (TRUE) {
     down(&customers);
     /* sleep if no customers */
     down(&mutex);
     --waiting;
     up(&barbers);
     up(&mutex);
     cut_hair();
   }
}
```

```
void customer (void)
{
   down(&mutex);
   if (waiting < CHAIRS) {
        ++waiting; /* sit down */
        up(&customers);
        up(&mutex);
        down(&barbers);
        get_haircut();
   } else { /* go elsewhere */
        up(&mutex);
   }
}
```

Semaphores:

- Customer – count waiting customers (excluding the one in the barber chair)
- Barbers – number of barbers who are idle
- mutex – for mutual exclusion

Coming up

- Deadlocks

  How deadlock arise and what you can do about them

Monitors in Java

```
public class ProducerConsumer {
        static final int N = 100;                     // constant giving the buffer size
        static producer p = new producer( );   // instantiate a new producer thread
        static consumer c = new consumer( );// instantiate a new consumer thread
        static our_monitor mon = new our_monitor( );  // instantiate a new monitor
        public static void main(String args[ ]) {
           p.start( );                                 // start the producer thread
           c.start( );                                 // start the consumer thread
        }
        static class producer extends Thread {
           public void run( )  {                       // run method contains the thread code
              int item;
              while (true) {                           // producer loop
                 item = produce_item( );
                 mon.insert(item);
              }
           }
           private int produce_item( ) { ... }    // actually produce
        }
        static class consumer extends Thread {
           public void run( )  {                       run method contains the thread code
              int item;
              while (true) {                           // consumer loop
                 item = mon.remove( );
                 consume_item (item);
              }
           }
           private void consume_item(int item) { ... }     // actually consume
        }
```

Monitors in Java

```
static class our_monitor {                    // this is a monitor
    private int buffer[ ] = new int[N];
    private int count = 0, lo = 0, hi = 0; // counters and indices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep( );    // if the buffer is full, go to sleep
        buffer [hi] = val;                 // insert an item into the buffer
        hi = (hi + 1) % N;                 // slot to place next item in
        count = count + 1;                 // one more item in the buffer now
        if (count == 1) notify( );         // if consumer was sleeping, wake it up
    }
    public synchronized int remove( ) {
        int val;
        if (count == 0) go_to_sleep( );    // if the buffer is empty, go to sleep
        val = buffer [lo];                 // fetch an item from the buffer
        lo = (lo + 1) % N;                 // slot to fetch next item from
        count = count – 1;                 // one few items in the buffer
        if (count == N – 1) notify( );     // if producer was sleeping, wake it up
        return val;
    }
   private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
}
}
```

Barriers

- To synchronize groups of processes
- Type of applications
  - Execution divided in phases
  - Process cannot go into new phase until all can
- e.g. Temperature propagation in a material