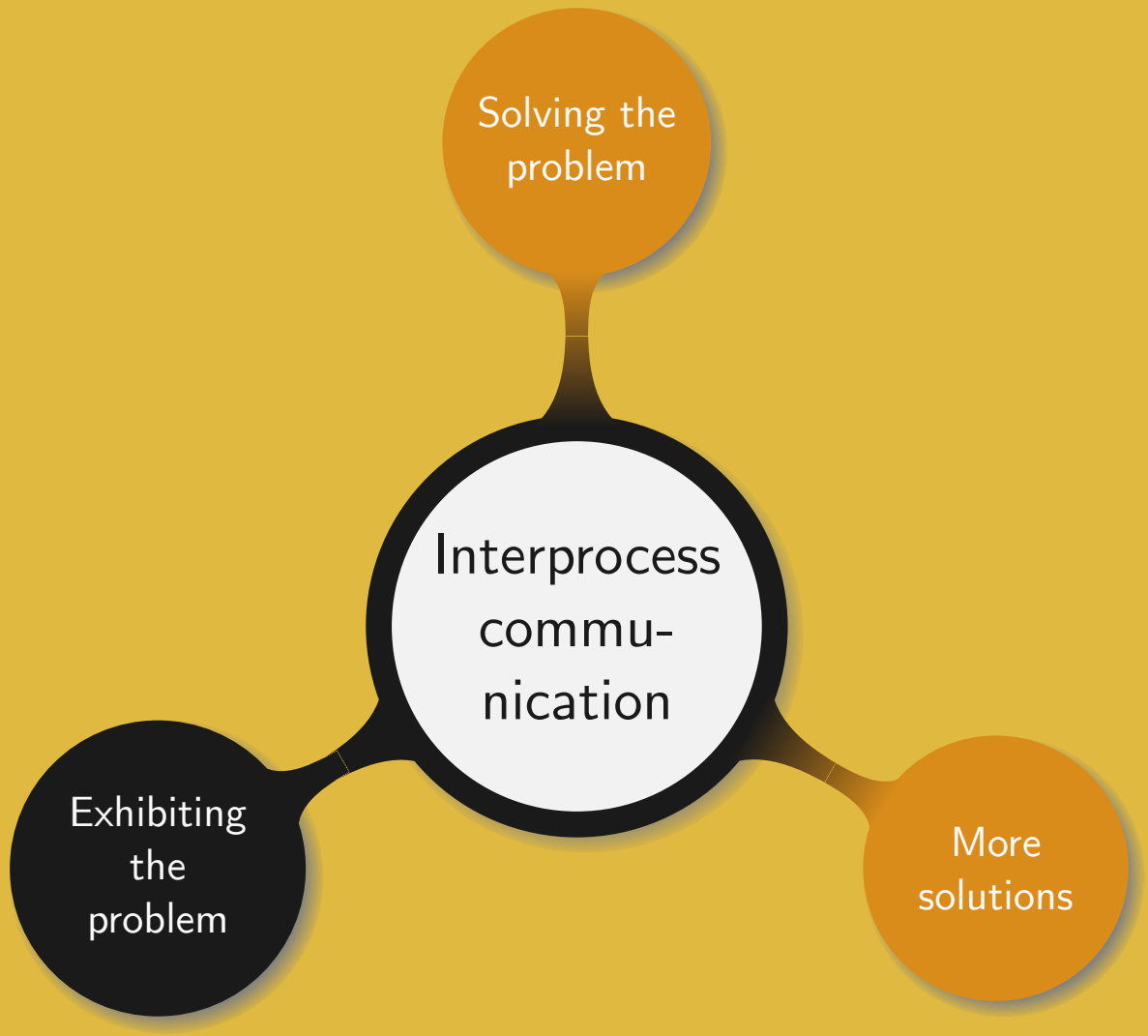
Introduction to Operating Systems

3. Interprocess communication

Manuel – Fall 2021

Chapter organisation



Working with a single thread

In single tasking all threads are independent:

- They cannot affect or be affected by anything
- Their state is not shared with other threads
- The input state determines the output
- Everything is reproducible
- Stopping or resuming does not lead to any side effect

It suffices to run a thread to completion and start the next one

Working with multiple threads

Difficulties appear with multi-tasking:

- A thread runs on one core at a time
- A thread can run on different cores at different times
- Each core is shared among several threads
- Several cores run several threads in parallel
- The number of cores has no impact on the running of the threads

Changes made by one thread can affect others

Thread and process cooperation

Setup for threads:

- Several threads share a common global variable
- The execution sequence impacts the global variable
- By default the behavior is random and irreproducible

Thread and process cooperation

Setup for threads:

- Several threads share a common global variable
- The execution sequence impacts the global variable
- By default the behavior is random and irreproducible

Major problems:

- How can threads share information?
- How to prevent them from getting on each other's way?
- How to ensure an acceptable running order?

Thread and process cooperation

Setup for threads:

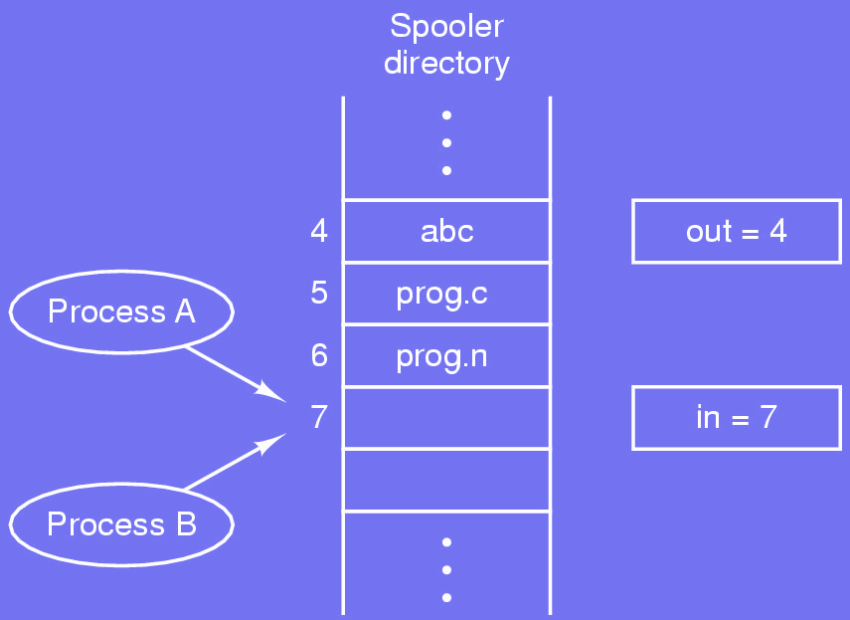
- Several threads share a common global variable
- The execution sequence impacts the global variable
- By default the behavior is random and irreproducible

Major problems:

- How can threads share information?
- How to prevent them from getting on each other's way?
- How to ensure an acceptable running order?

All those thread issues within a process can be extended to processes within the operating system

Race conditions



In the printing spool:

1. *A* wants to queue a file: reads `next_free_slot=7`
2. An interrupt occurs
3. *B* wants to queue a file: reads `next_free_slot=7`
4. *B* queues its file in slot 7, and updates `next_free_slot=8`
5. *A* queues its file in slot 7

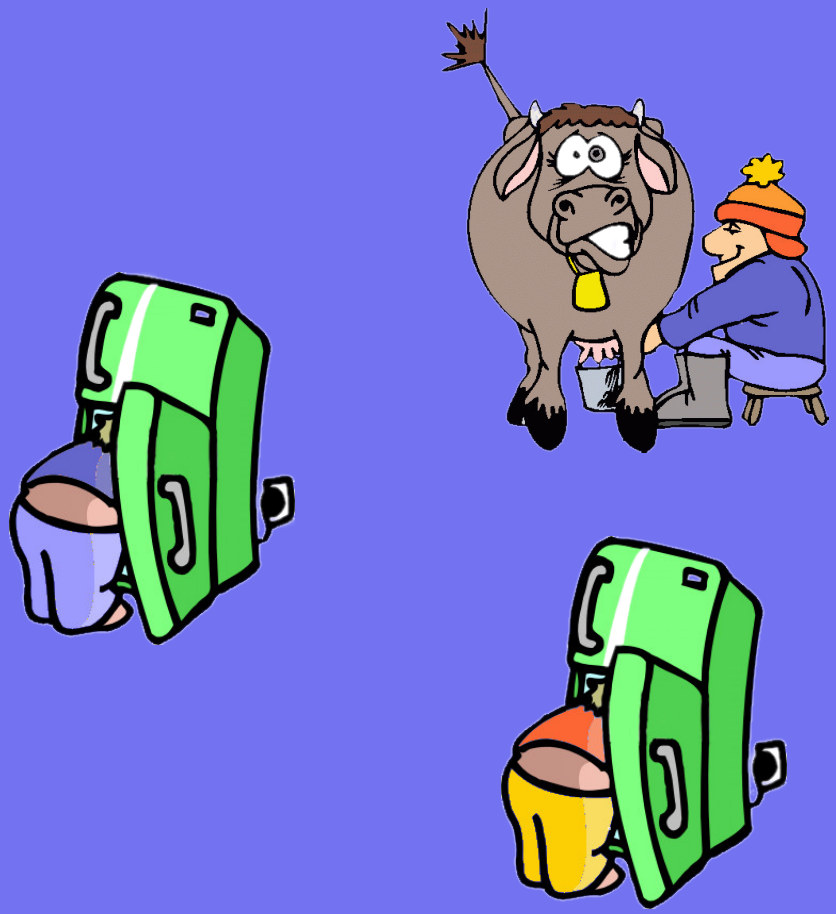
Too much milk

9:00 am	9:15 am	9:30 am	9:45 am

Too much milk

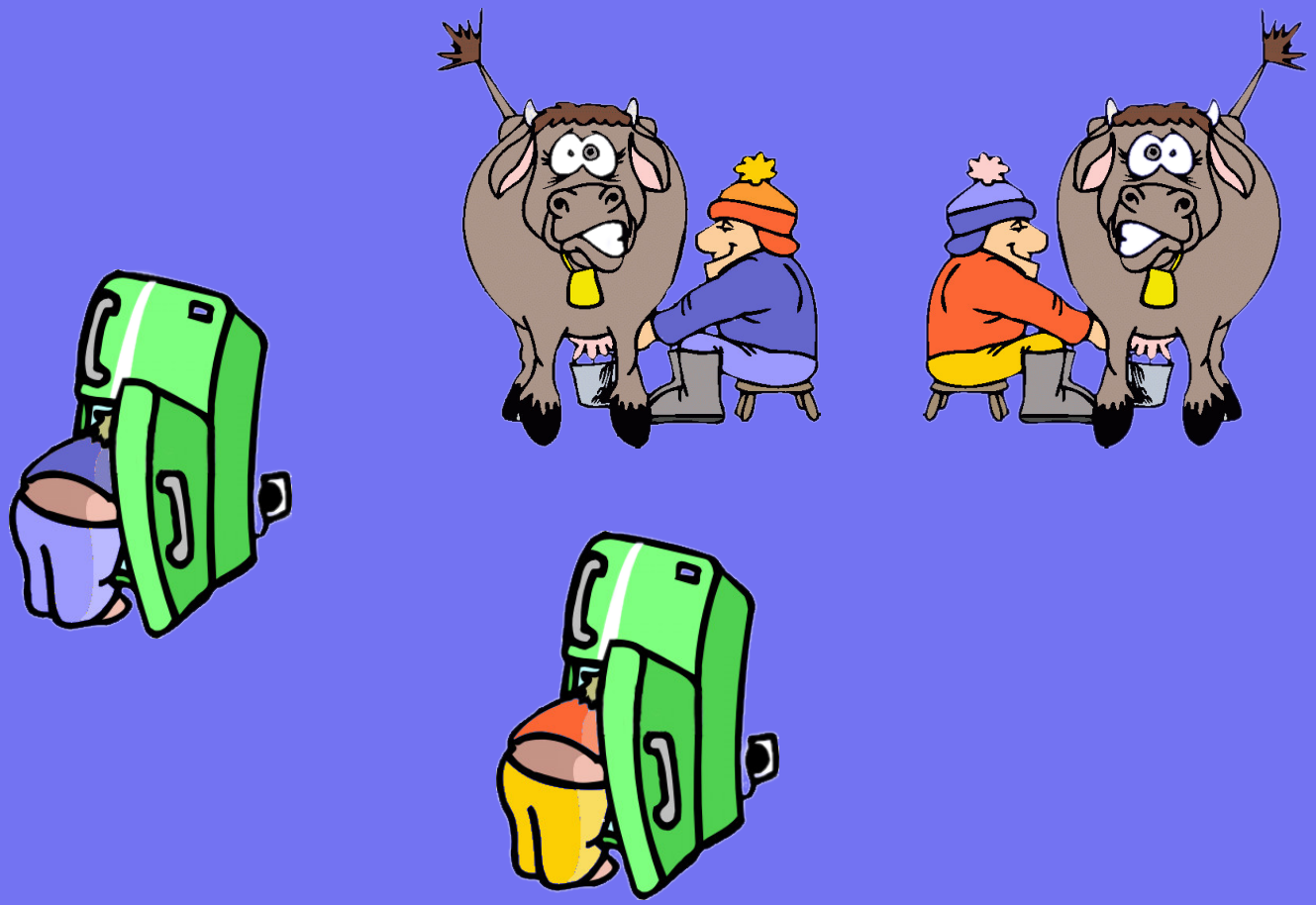
9:00 am	9:15 am	9:30 am	9:45 am

Too much milk



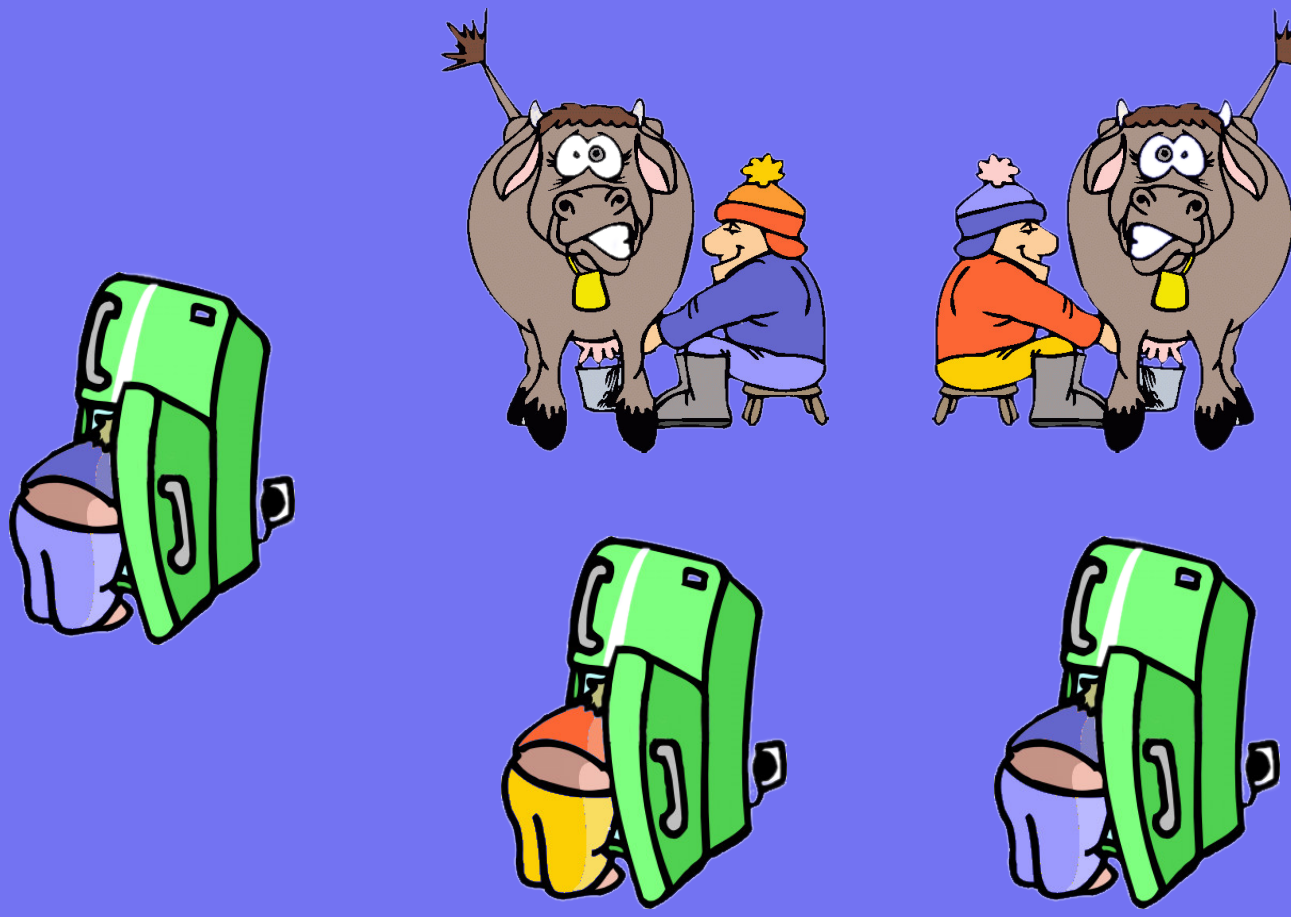
9:00 am	9:15 am	9:30 am	9:45 am

Too much milk



9:00 am 9:15 am 9:30 am 9:45 am

Too much milk

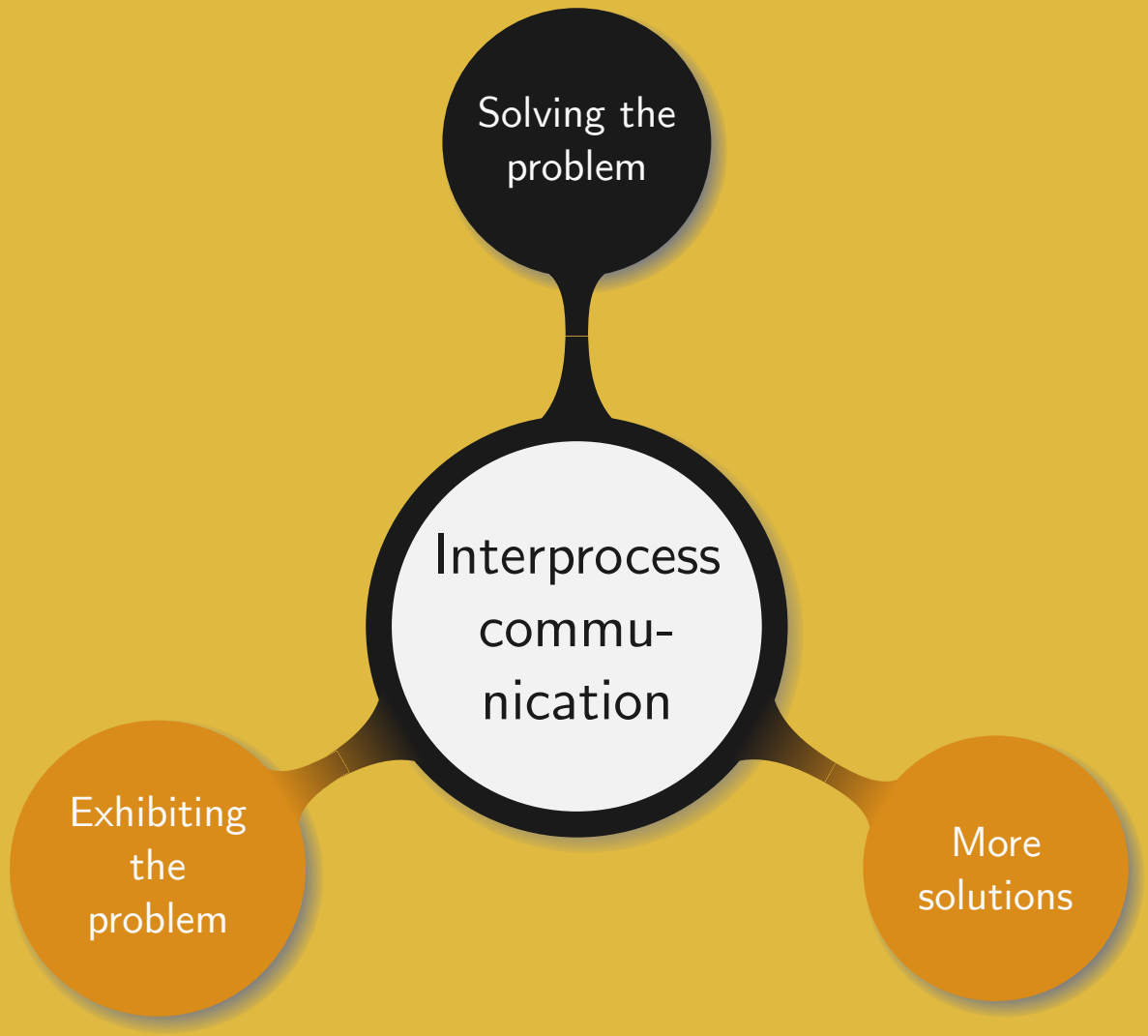


9:00 am 9:15 am 9:30 am 9:45 am

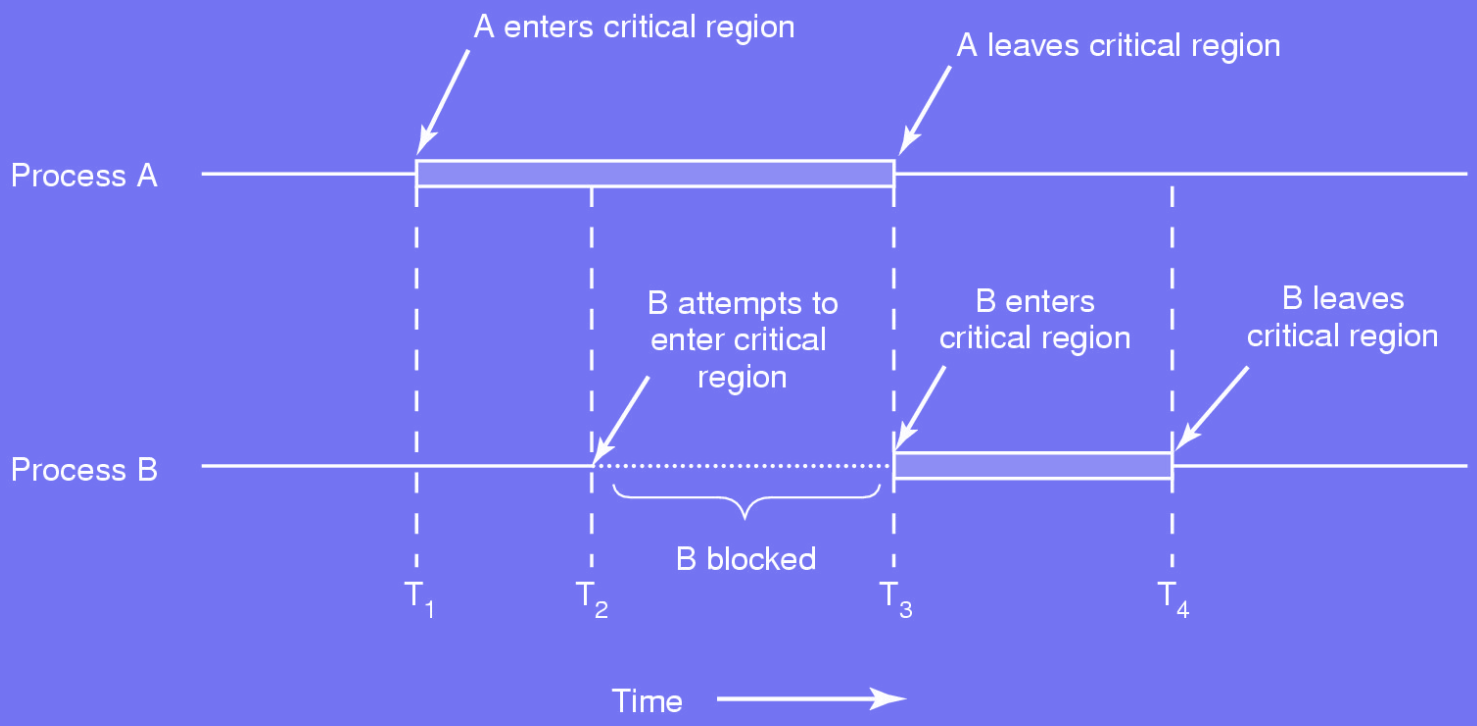
Too much milk

9:00 am 9:15 am 9:30 am 9:45 am

Chapter organisation



Critical region



Part of the program where shared memory is accessed:

- No two processes can be in a critical region at a same time
- No assumption on the speed or number of CPUs
- No process outside a critical region can block other processes
- No process waits forever to enter a critical region

Too much milk

Frank

```
1  if(no milk && no note) {
2    leave note;
3    milk the cow;
4    remove note;
5  }
```

John

```
1  if(no milk && no note) {
2    leave note;
3    milk the cow;
4    remove note;
5  }
```

Too much milk

Frank

```
1  if(no milk && no note) {
2    leave note;
3    milk the cow;
4    remove note;
5  }
```

```
1  leave note Frank;
2  if(no note John) {
3    if(no milk) milk the cow;
4  }
5  remove note Frank;
```

John

```
1  if(no milk && no note) {
2    leave note;
3    milk the cow;
4    remove note;
5  }
```

```
1  leave note John;
2  if(no note Frank) {
3    if(no milk) milk the cow;
4  }
5  remove note John;
```

What is the issue with those two strategies?

Too much milk

Frank

```
1  leave note Frank;
2  while(note John) {
3    nothing;
4  }
5  if(no milk) {
6    milk the cow;
7  }
8  remove note Frank;
```

John

```
1  leave note John;
2  if(no note Frank) {
3    if(no milk) {
4      milk the cow;
5    }
6  }
7  remove note John;
```

How good is this strategy?

Peterson's idea

Symmetric strategy for two processes:

- When wanting to enter a critical region a process:
  - Shows its interest for the critical region
  - If it is accessible it exits the function and accesses it
  - If it is not accessible it waits in a tight loop
- When a process has completed its work in the critical region it signals its departure

What is the main drawback of this strategy?

Peterson's idea

Pseudo C code for two processes represented as 0 and 1

```
1  #define TRUE 1
2  #define FALSE 0
3  int turn;
4  int interested[2];
5  void enter_region(int p) {
6    int other;
7    other=1-p;
8    interested[p]=TRUE;
9    turn=p;
10   while(turn==p && interested[other]==TRUE)
11 }
12 void leave_region(int p) {
13   interested[p]=FALSE;
14 }
```

Peterson's idea

Side effects of Peterson's idea:

- Two processes: L, low priority, and H, high priority
- L enters in a critical region
- H becomes ready
- H has higher priority so the scheduler switches to H
- L has lower priority so is not rescheduled as long as H is busy
- H loops forever

Mutual exclusion at hardware level

Prevent the process in the critical region from being stopped:

- Disable interrupts:
  - Can be done within the kernel for a few instructions
  - Cannot be done by user processes
  - Only works when there is a single CPU
  - An interrupt on another CPU can still mess up the shared variable
- Use atomic operations:
  - Either happens in its entirety or not at all
  - Several operations can be performed at once, e.g. $A = B$
  - Requires the CPU to support the atomic update of a memory space
  - Can be used to prevent other CPUs to access a shared memory

Using the TSL instruction

A simple atomic operation:

- Test and Set Lock: TSL
- Copies LOCK to a register and set it to 1
- LOCK is used to coordinate the access to a shared memory
- Ensures LOCK remains unchanged while checking its value

```
1  enter_region:
2    TSL REGISTER,LOCK
3    CMP REGISTER,#0
4    JNE enter_region
5    RET
6
7  leave_region:
8    MOVE LOCK,#0
9    RET
```

Using the TSL instruction

A simple atomic operation:

- Test and Set Lock: TSL
- Copies LOCK to a register and set it to 1
- LOCK is used to coordinate the access to a shared memory
- Ensures LOCK remains unchanged while checking its value

```
1  enter_region:
2    TSL REGISTER,LOCK
3    CMP REGISTER,#0
4    JNE enter_region
5    RET
6
7  leave_region:
8    MOVE LOCK,#0
9    RET
```

Is this strategy better than Peterson's idea?

Consumer-producer problem

```
1  #define N 100
2  int count=0;
3  void producer() {
4    int item;
5    while(1) {
6      item=produce_item(); if(count==N) sleep();
7      insert_item(item); count++;
8      if(count==1) wakeup(consumer);
9    }
10 }
11 void consumer() {
12   int item;
13   while(1) {
14     if(count==0) sleep();
15     item=remove_item(); count--;
16     if(count==N-1) wakeup(producer); consume_item(item);
17   }
18 }
```

Is this code exhibiting any problem?

Consumer-producer problem

Assume the buffer is empty:

- Consumer reads `count == 0`
- Scheduler stops the consumer and starts the producer
- Producer adds one item
- Producer wakes up the consumer
- Consumer not yet asleep, signal is lost
- Consumer goes asleep
- When the buffer is full the producer falls asleep
- Both consumer and producer sleep forever

Semaphore

Basics:

- Introduced by Dijkstra in 1965
- Simple hardware based solution
- Basis of all modern OS synchronization mechanisms

Semaphore

Basics:

- Introduced by Dijkstra in 1965
- Simple hardware based solution
- Basis of all modern OS synchronization mechanisms

A semaphore `sem` is:

- A positive integer variable
- Only changed or tested through two actions

```
1 down(sem) {
2   while(sem==0) sleep();
3   sem--;
4 }
```

```
1 up(sem) {
2   sem++;
3 }
```

Semaphores and processes

The `down` operation

- If $sem > 0$, decrease it and continue
- If $sem = 0$, sleep and do not complete the down

The `up` operation

- Increment the value of the semaphore
- An awaken sleeping process can complete its down

Semaphores and processes

The `down` operation

- If $sem > 0$, decrease it and continue
- If $sem = 0$, sleep and do not complete the down

The `up` operation

- Increment the value of the semaphore
- An awaken sleeping process can complete its down

Checking or changing the value and sleeping are done atomically:

- Single CPU: disable interrupts
- Multiple CPUs: use `TSL` to ensure only one CPU accesses the semaphore

Is disabling the interrupts to process the semaphore an issue?

Semaphores and interrupts

Using semaphores to hide interrupts:

- Each IO device gets a semaphore initialised to 0
- A process accessing the device applies a `down`
- The process becomes blocked
- An interrupt is issued when the device has completed the work
- The interrupt handler processes the interrupt and applies an `up`
- The process becomes ready

Mutexes

A mutex is a semaphore taking values 0 (unlocked) or 1 (locked)

On a mutex-lock request:

- If the mutex is unlocked:
  - Lock the mutex
  - Enter the critical region
- If mutex is locked: put the calling thread asleep
- When the thread in the critical region exits:
  - Unlock the mutex
  - Allow a thread to acquire the lock and enter the critical region

Mutex implementation

Mutexes can be implemented in user-space using `TSL`

```
1  mutex-lock:
2    TSL REGISTER,MUTEX
3    CMP REGISTER,#0
4    JZ ok
5    CALL thread_yield
6    JMP mutex-lock
7  ok: RET
8
9  mutex-unlock:
10   MOVE MUTEX,#0
11   RET
```

Questions:

- What differences were introduced compared to `enter_region` (3.16)?
- In user-space what happens if a thread tries to acquire lock through busy-waiting?
- Why is `thread_yield` used?

Consumer-producer problem

consumer_producer.c

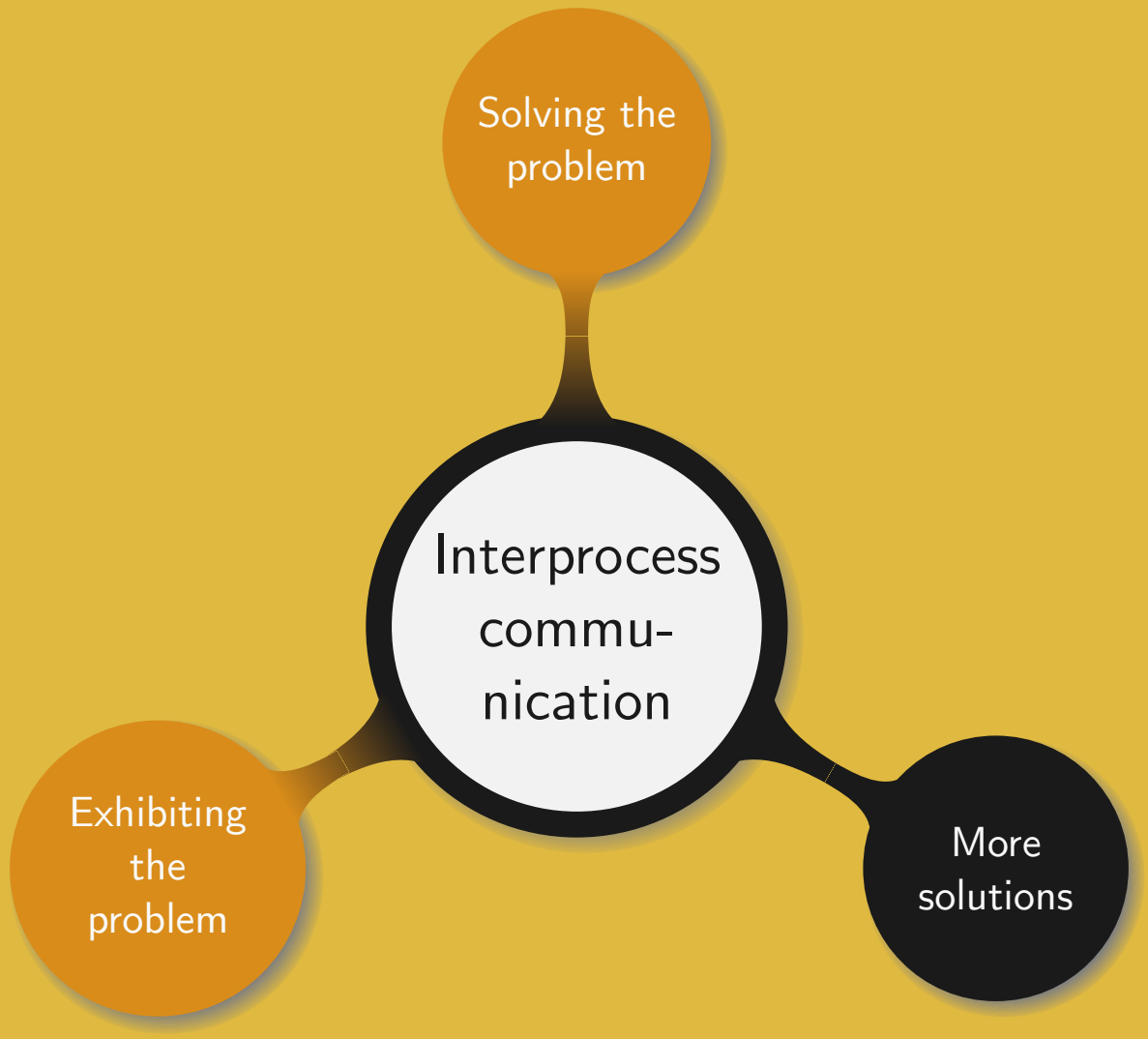
```
1  #include <stdio.h>
2  #include <pthread.h>
3  #define MAX 1000
4  pthread_mutex_t m; pthread_cond_t cc, cp; int buf=0;
5  void *prod() {
6    for(int i=1;i<MAX;i++) {
7      pthread_mutex_lock(&m); while(buf!=0) pthread_cond_wait(&cp,&m);
8      buf=1; pthread_cond_signal(&cc); pthread_mutex_unlock(&m);
9    }
10   pthread_exit(0);
11 }
12 void *cons() {
13   for(int i=1;i<MAX;i++) {
14     pthread_mutex_lock(&m);  while(buf==0) pthread_cond_wait(&cc,&m);
15     buf=0; pthread_cond_signal(&cp); pthread_mutex_unlock(&m);
16   }
17   pthread_exit(0);
18 }
19 int main() {
20   pthread_t p, c;
21   pthread_mutex_init(&m,0);  pthread_cond_init(&cc,0); pthread_cond_init(&cp,0);
22   pthread_create(&c,0,cons,0); pthread_create(&p,0,prod,0);
23   pthread_join(p,0); pthread_join(c,0);
24   pthread_cond_destroy(&cc); pthread_cond_destroy(&cp); pthread_mutex_destroy(&m);
25 }
```

Consumer-producer problem

Alter the previous program such as:

- To display information on the consumer and producer
- To increase the buffers to 100
- To have two consumers and one producer. In this case also print which consumer is active.

Chapter organisation



Consumer-producer problem

```
1  mutex mut = 0; semaphore empty = 100; semaphore full = 0;
2  void producer() {
3    while(TRUE) {
4      item = produce_item();
5      mutex-lock(&mut);
6      down(&empty); insert_item(item);
7      mutex-unlock(&mut);
8      up(&full);
9    }
10 }
11 void consumer() {
12   while(TRUE) {
13     down(&full);
14     mutex-lock(&mut); item = remove_item(); mutex-unlock(&mut);
15     up(&empty); consume_item(item);
16   }
17 }
```

Is this code working as expected?

Consumer-producer problem

In the previous code:

- What is the behavior of the producer when the buffer is full?
- What about the consumer?
- What is the final result for this program?
- How to fix it?

Monitors

Monitors are an attempt to merge synchronization with OOP

Basic idea behind monitors:

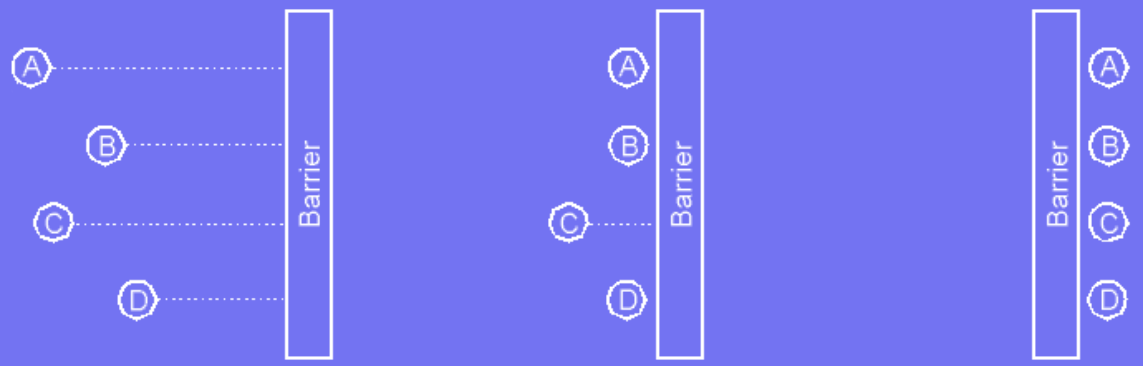
- Programming concept that must be known by the compiler
- The mutual exclusion is not handled by the programmer
- Locking occurs automatically
- Only one process can be active within a monitor at a time
- A monitor can be seen as a "special type of class"
- Processes can be blocked and awaken based on condition variables and `wait` and `signal` functions

Consumer-producer problem

```
1  monitor ProducerConsumer {
2    condition full, empty;
3    int count;
4    void insert(item) {
5      if (count == N) wait(full);
6      insert_item(item);
7      count++;
8      if (count==1) signal(empty);
9    }
10   void remove() {
11     if (count==0) wait(empty);
12     removed = remove_item;
13     count--;
14     if (count==N-1) signal(full);
15   }
16   count:= 0;
17 }
```

```
1  void ProducerConsumer::producer() {
2    while (TRUE) {
3      item = produce_item();
4      ProducerConsumer.insert(item);
5    }
6  }
7  void ProducerConsumer::consumer() {
8    while (TRUE) {
9      item=ProducerConsumer.remove();
10     consume_item(item)
11   }
12 }
```

Barriers



Useful when several processes must complete before the next phase

Key points

- Why is thread communication essential?
- What is a critical region?
- Do software solutions exist?
- What is an atomic operation?
- What are the two best and most common solutions?

Thank you!