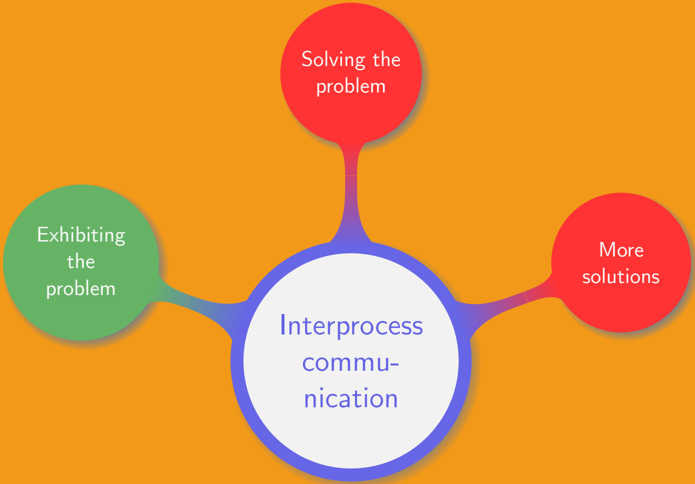
Introduction to Operating Systems

3. Interprocess communication

Manuel – Fall 2020

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:

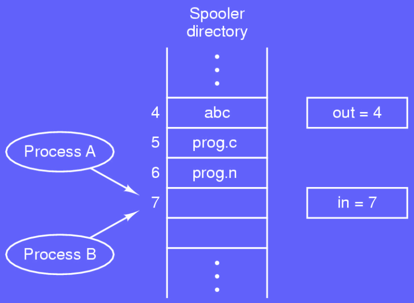
- Several threads share a common global variable
- The execution sequence impacts the global variable
- By default the behavior is random and irreproducible

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

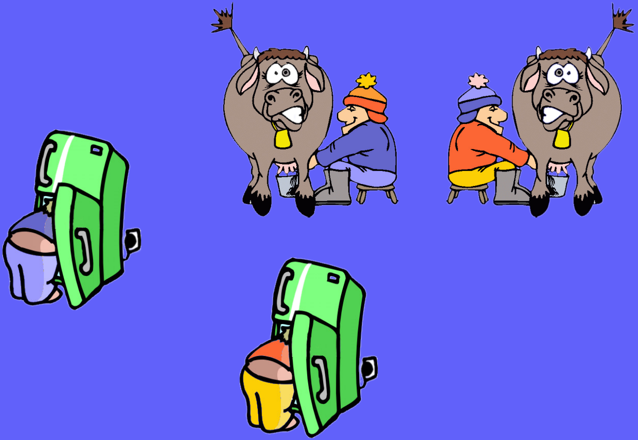
Too much milk

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

Too much milk

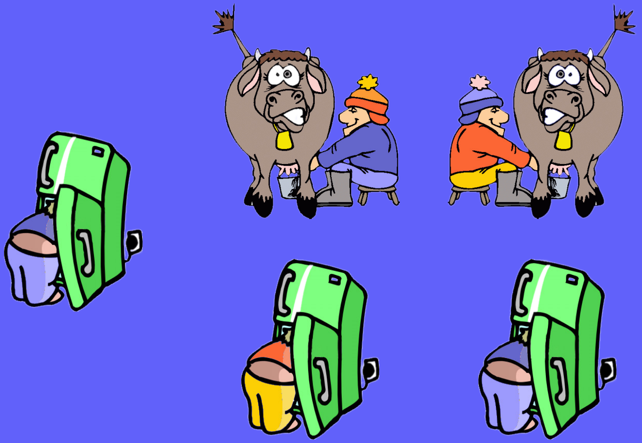
9:00 am | 9:15 am | 9:30 am | 9:45 am

Too much milk



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

Too much milk

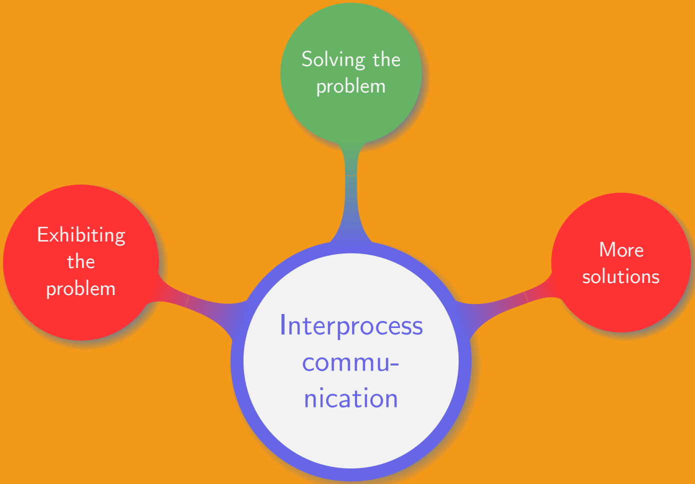


9:00 am	9:15 am	9:30 am	9:45 am

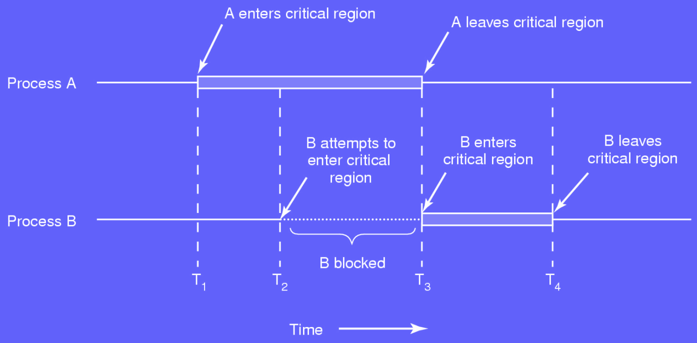
Too much milk

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

```
if(no milk && no note) {
  leave note;
  milk the cow;
  remove note;
}
```

John

```
if(no milk && no note) {
  leave note;
  milk the cow;
  remove note;
}
```

Too much milk

Frank

```
if(no milk && no note) {
  leave note;
  milk the cow;
  remove note;
}
```

John

```
if(no milk && no note) {
  leave note;
  milk the cow;
  remove note;
}
```

```
leave note Frank;
if(no note John) {
  if(no milk) milk the cow;
}
remove note Frank;
```

```
leave note John;
if(no note Frank) {
  if(no milk) milk the cow;
}
remove note John;
```

What is the issue with those two strategies?

Too much milk

Frank

```
leave note Frank;
while(note John) {
  nothing;
}
if(no milk) {
  milk the cow;
}
remove note Frank;
```

John

```
leave note John;
if(no note Frank) {
  if(no milk) {
    milk the cow;
  }
}
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

```c
#define TRUE 1
#define FALSE 0
int turn;
int interested[2];
void enter_region(int p) {
  int other;
  other=1-p;
  interested[p]=TRUE;
  turn=p;
  while(turn==p && interested[other]==TRUE)
}
void leave_region(int p) {
  interested[p]=FALSE;
}
```

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

```
enter_region:
  TSL REGISTER,LOCK
  CMP REGISTER,#0
  JNE enter_region
  RET

leave_region:
  MOVE LOCK,#0
  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

```
enter_region:
  TSL REGISTER,LOCK
  CMP REGISTER,#0
  JNE enter_region
  RET

leave_region:
  MOVE LOCK,#0
  RET
```

Is this strategy better than Peterson's idea?

Consumer-producer problem

```c
#define N 100
int count=0;
void producer() {
  int item;
  while(1) {
    item=produce_item(); if(count==N) sleep();
    insert_item(item); count++;
    if(count==1) wakeup(consumer);
  }
}
void consumer() {
  int item;
  while(1) {
    if(count==0) sleep();
    item=remove_item(); count--;
    if(count==N-1) wakeup(producer); consume_item(item);
  }
}
```

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

```
down(sem) {
  while(sem==0) sleep();
  sem--;
}
```

```
up(sem) {
  sem++;
}
```

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 I/O 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`

```
mutex-lock:
  TSL REGISTER,MUTEX
  CMP REGISTER,#0
  JZ ok
  CALL thread_yield
  JMP mutex-lock
ok: RET

mutex-unlock:
  MOVE MUTEX,#0
  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

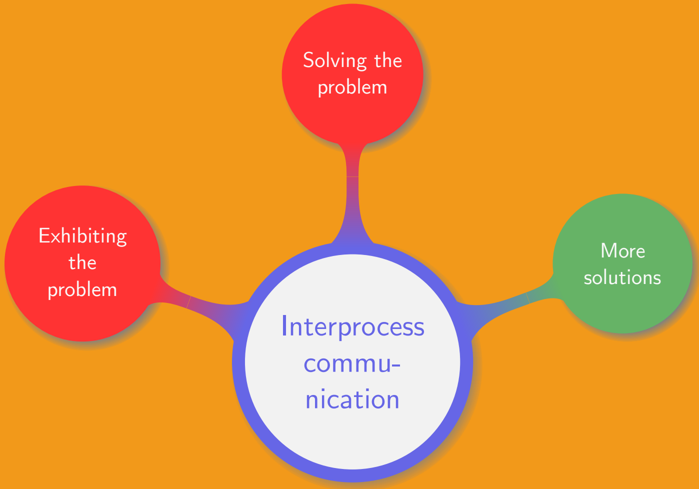
```c
#include <stdio.h>
#include <pthread.h>
#define MAX 1000
pthread_mutex_t m; pthread_cond_t cc, cp; int buf=0;
void *prod() {
  for(int i=1;i<MAX;i++) {
    pthread_mutex_lock(&m); while(buf!=0) pthread_cond_wait(&cp,&m);
    buf=1; pthread_cond_signal(&cc); pthread_mutex_unlock(&m);
  }
  pthread_exit(0);
}
void *cons() {
  for(int i=1;i<MAX;i++) {
    pthread_mutex_lock(&m);  while(buf==0) pthread_cond_wait(&cc,&m);
    buf=0; pthread_cond_signal(&cp); pthread_mutex_unlock(&m);
  }
  pthread_exit(0);
}
int main() {
  pthread_t p, c;
  pthread_mutex_init(&m,0);  pthread_cond_init(&cc,0); pthread_cond_init(&cp,0);
  pthread_create(&c,0,cons,0); pthread_create(&p,0,prod,0);
  pthread_join(p,0); pthread_join(c,0);
  pthread_cond_destroy(&cc); pthread_cond_destroy(&cp); pthread_mutex_destroy(&m);
}
```

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

```
mutex mut = 0; semaphore empty = 100; semaphore full = 0;
void producer() {
  while(TRUE) {
    item = produce_item();
    mutex-lock(&mut);
    down(&empty); insert_item(item);
    mutex-unlock(&mut);
    up(&full);
  }
}
void consumer() {
  while(TRUE) {
    down(&full);
    mutex-lock(&mut); item = remove_item(); mutex-unlock(&mut);
    up(&empty); consume_item(item);
  }
}
```

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

```
monitor ProducerConsumer {
  condition full, empty;
  int count;
  void insert(item) {
    if (count == N) wait(full);
    insert_item(item);
    count++;
    if (count==1) signal(empty);
  }
  void remove() {
    if (count==0) wait(empty);
    removed = remove_item;
    count--;
    if (count==N-1) signal(full);
  }
  count:= 0;
}
```

```
void ProducerConsumer::producer() {
  while (TRUE) {
    item = produce_item();
    ProducerConsumer.insert(item);
  }
}
void ProducerConsumer::consumer() {
  while (TRUE) {
    item=ProducerConsumer.remove();
    consume_item(item)
  }
}
```

Message passing

Over a distributed system semaphores and mutexes cannot be use

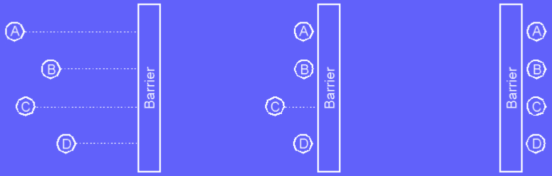
Message passing strategy:

- `send(destination,&message)`
- `receive(source,&message)`: blocks or exit if nothing is received

Potential issues:

- Messages can get lost, e.g. sending or acknowledging reception
- Confusion on the process names
- Security, e.g. authentication, traffic encryption
- Performance

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!