Introduction to Operating Systems

Chapter 3: Interprocess communication

Manuel

Fall 2017

Outline

1. Exhibiting the problem
2. Solving the problem
3. More solutions

Easy setup

Independent threads:

- Cannot affect/be affected by anything
- State not shared with other threads
- Input state determines the output
- Reproducible
- No side effect when stopping/resuming

Executing threads

Where difficulty starts:

- Single-tasking: run a thread to completion and start next one
- Multi-tasking:
  - One core shared among several threads
  - Several cores run several threads in parallel
- A thread runs on one core at a time
- A thread can run on different cores at different imes
- For threads no difference between one or more cores

Cooperating threads

Setup:

- Threads sharing a state
- Behavior depends on the execution sequence
- Behavior may seem random/irreproducible

Cooperating threads

Setup:

- Threads sharing a state
- Behavior depends on the execution sequence
- Behavior may seem random/irreproducible

Major problems:

1. How can threads/processes share information?
2. How to prevent them for getting in each other's way?
3. How to handle sequencing?

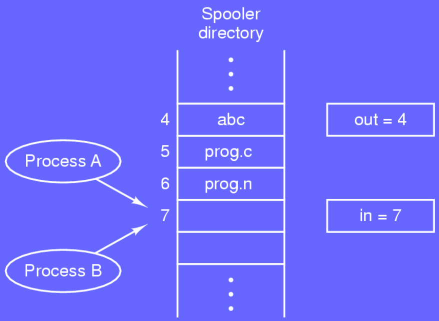
Atomic operation

Operation that either happens in its entirety or not at all

- Atomic operation cannot be created
- Atomic operations are hardware level
- Central to the question of parallel programming

Example of atomic operation: A=B, read a clean value for B and sets a clean value for A

Race conditions



- Process A wants to queue a file, reads next_free_slot=7
- Interrupt occurs, Process B reads next_free_slot=7
- Process B queues its file in slot 7, and update next_free_slot=8
- Process A resumes using next_free_slot=7
- Game over

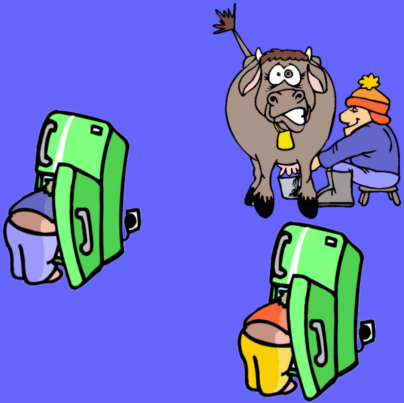
Too much milk

9AM 9:15AM 9:30AM 9:45AM

Too much milk

9AM 9:15AM 9:30AM 9:45AM

Too much milk

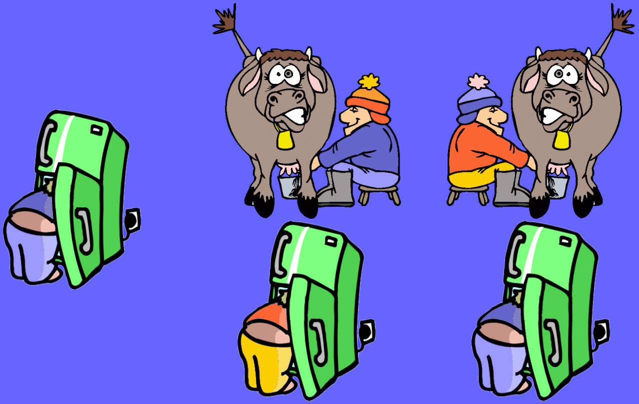


9AM	9:15AM	9:30AM	9:45AM

Too much milk

9AM	9:15AM	9:30AM	9:45AM

Too much milk



9AM 9:15AM 9:30AM 9:45AM

Too much milk

9AM 9:15AM 9:30AM 9:45AM

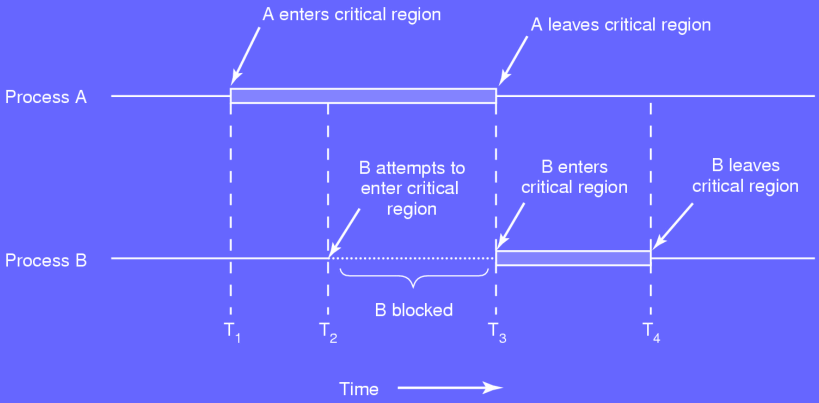
Outline

Critical region

Part of the program where shared memory is accessed:

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

Mutual exclusion



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;
}
```

Result: too much milk

Too much milk

Frank

```
leave note Frank;
if(no note John) {
  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;
```

Too much milk

Frank

```
leave note Frank;
if(no note John) {
  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;
```

Result: no milk

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;
```

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;
```

Result: just enough milk

Too much milk

Comments on the solution:

- Solution is correct
- Complicated
- Asymmetrical
- Busy wait, CPU time wasted

Peterson's idea

Simple strategy:

- Assume two processes
- Two functions:
  - `enter_region`: show process is interested and wait for its turn
  - `leave_region`: indicates departure from critical region
- Solution is correct
- Drawback: busy wait

Peterson's idea

Simple strategy:

- Assume two processes
- Two functions:
  - `enter_region`: show process is interested and wait for its turn
  - `leave_region`: indicates departure from critical region
- Solution is correct
- Drawback: busy wait

Exercise: "implement" Peterson's idea (use two processes represented by the integers 0 and 1)

Peterson's idea

```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

Software vs. hardware?

Disabling interrupts, good or bad?

Software vs. hardware?

Disabling interrupts, good or bad?

- Case 1: block interrupts for the whole computer
  - Interrupts could be disabled for a while (too long?)
  - This gives a lot of power to the programmer
- Case 2: block interrupts for only one CPU
  - No effect on other processors
  - Another CPU can access the variable between read and write

The TSL instruction

- Test and Set Lock (TSL)
- Hardware level, requires assembly
- Task: copy *lock* to register and set *lock* to 1
- Atomic operation

The TSL instruction

- Test and Set Lock (TSL)
- Hardware level, requires assembly
- Task: copy *lock* to register and set *lock* to 1
- Atomic operation

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

  leave_region:
    MOVE LOCK,#0
    RET
```

The TSL instruction

- Test and Set Lock (TSL)
- Hardware level, requires assembly
- Task: copy *lock* to register and set *lock* to 1
- Atomic operation

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

  leave_region:
    MOVE LOCK,#0
    RET
```

Note: TSL displays the same side-effect as Peterson's solution when dealing with processes having different priorities

Blocking strategy

```
#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);
  }
}
```

Race condition

Setup: 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

Baiscs:

- 1965, Edseger Dijkstra
- Simple hardware based solution
- Basis of all contemporary OS synchronization mechanisms

Semaphore

Baiscs:

- 1965, Edseger Dijkstra
- Simple hardware based solution
- Basis of all contemporary OS synchronization mechanisms

A semaphore *s* in a non-negative variable that can only be changed or tested by two atomic actions:

```
down(s) {
  while(s==0) wait();
  s--;
}
```

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

Semaphore and processes

The down operation

Check if the value is > 0

- True: decrement the value and continues
- False: sleep, do not complete the down

Only one single atomic action

The up operation

- Increment the value of the semaphore
- If one or more processes were asleep, randomly choose one to wakeup (complete its down)

Only one single atomic action

Semaphore implementation

Semaphores MUST be implemented in an indivisible way

- Up and down are implemented using system calls
- OS disables all interrupts while testing, updating the semaphore and putting process to sleep
- When dealing with more than one CPU, semaphores are protected using the TSL instruction

Note: a semaphore operation only takes a few microseconds

Semaphore and interrupts

Hiding interrupts using semaphores:

- Each I/O device gets a semaphore initialised to 0
- Managing process applies a `down` when starting an I/O device
- Process is blocked
- Interrupt handler applies an `up` when receiving an interrupt
- Process is ready to run again

Mutex

MUTual EXclusion:

- Simplified semaphore
- Takes values 0 (unlocked) or 1 (locked)

On a mutex-lock request:

- If mutex is currently unlocked, lock it; thread can enter in critical region
- If mutex is currently locked, block the calling thread until thread in critical regions is done and calls mutex-unlock
- Randomly chose which thread acquires the lock if more than one are waiting

Mutex implementation

Mutexes can be implemented in user-space using TSL

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

mutex-unlock:
    MOVE MUTEX,#0
    RET
```

- What differences were introduced compared to `enter_region` (3.20)?
- In user-space what happens if a thread tries to acquire lock through busy-waiting?
- Why is `thread_yield` used?

Example

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(void *p) {
  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(void *p) {
  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);
}
```

Example

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.

Outline

Dangers. . .

```
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);
  }
}
```

Questions

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

Semaphores and mutexes are convenient but sometimes risky

The concept of monitor:

- Monitors are higher level, cleaner and less risky solution
- A monitor is composed of
  - Procedures/functions
  - Structures
- Only one process can be active within a monitor at a time
- Monitors are a programming concept, compiler must know them
- Mutual exclusion handled by the compiler not the programmer

Monitors

Basics on monitors:

- Monitors are easy to use for mutual exclusion
- Offer a condition to block “properly” when the buffer is full
- A signal on the condition variable can wake up a blocked process
- Only one process can be active in the monitor
- As soon as the signal on the condition is sent, exit the monitor
- Other process can resume

Example

```
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);
    return(removed);
  }
  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

Limitation of semaphore/monitor: processes need to share some part of the memory. Distributed systems over a network consist of multiple CPU each with its own private memory

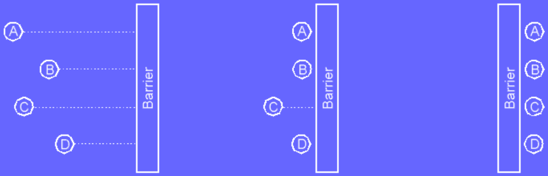
Message passing:

- `send(destination,&message)`
- `receive(source,&message)`, can either block or exit if nothing is received

Potential issues:

- Message lost (sending/acknowledging reception)
- No possible confusion on process names
- Security (authentication, traffic)
- Performance

Barriers



Useful for programs/problems where all processes must finish before going to 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!