

Chapter V

INTER-PROCESS SYNCHRONIZATION

Jehan-François Pâris
jfparis@uh.edu

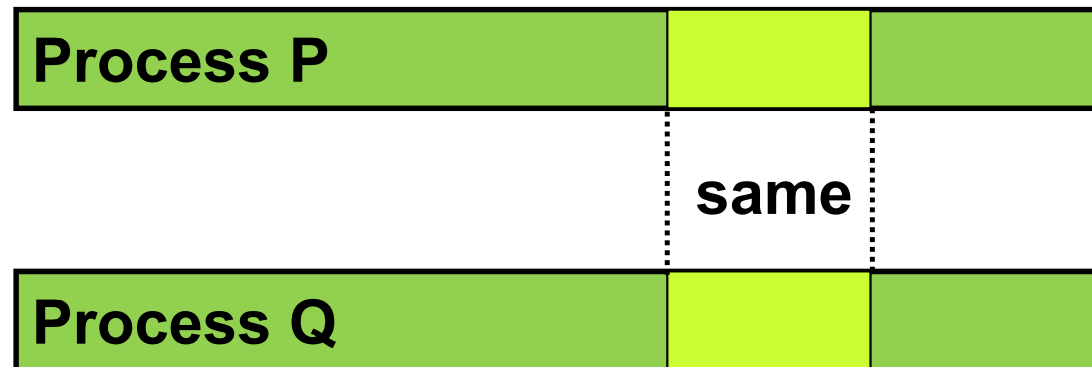


Chapter overview

- ***The critical section problem***
- ***Spin locks***
 - Intel realization
- ***Semaphores***
- ***Classical synchronization problems***
- ***Monitors***
- ***Advanced pthread synchronization***

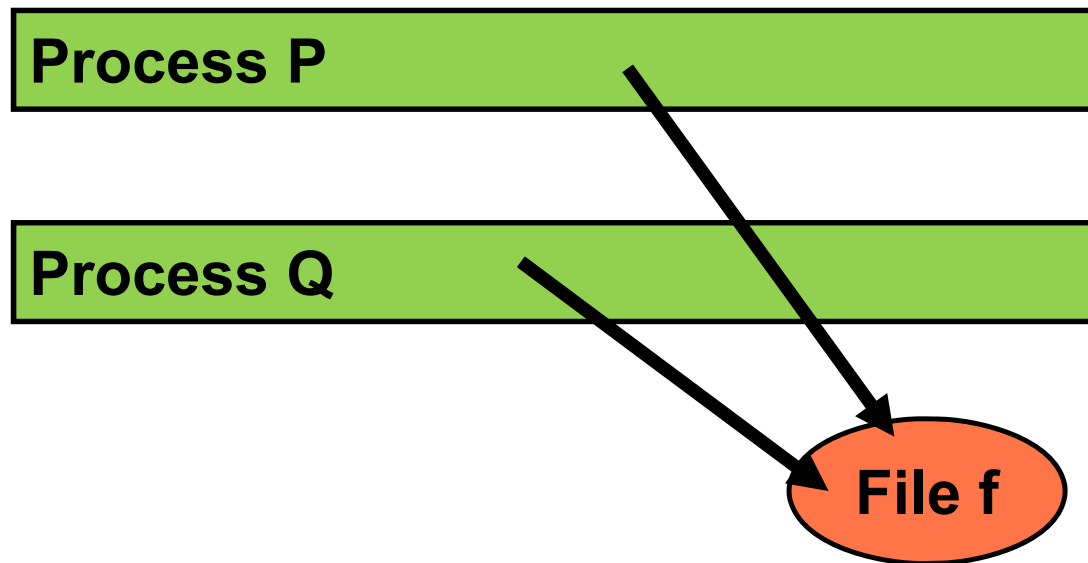
Shared Memory

- Name says it
 - Two or more processes share a part of their address space



Shared files

- Two processes access the same file at the same time





The outcome

- We can expect incorrect results whenever two processes (or two threads of a process) modify the same data at the same time



First example (I)

■ Bank Account Management

- Each account has a balance
- One process tries to ***credit*** an account
- Another process tries to ***debit*** the same account



The credit function

```
do_credit(int account, float amount) {  
    float balance;  
    fp = fopen(...); // open file  
    fseek(fp, ...); // locate account  
    A fscanf(fp,"%f", &balance); // read balance  
    balance += amount; // update balance  
    fseek(fp, ...); // go back to account  
    B fprintf(fp,"%f", balance); // save result  
} // do_credit
```



The debit function

```
do_debit(int account, float amount) {  
    float balance;  
    fp = fopen(...); // open file  
    fseek(fp, ...); // locate account  
    C fscanf(fp, "%f", &balance); // read balance  
    balance -= amount; // update balance  
    fseek(fp, ...); // go back to account  
    D fprintf(fp, "%f", balance); // save result  
} // do_debit
```




First example (II)

- Assuming
 - Balance is \$500
 - Amount to be credited is \$3000
 - Amount to be debited is \$100
- Sole correct value for final balance is \$3400
- We can also get **\$3500** and **\$400!**
 - *Everything depends on ordering of reads and writes*



Explanation (I)

- Correct result is guaranteed if the two processes access the data one after the other
 - I/Os are done in the order
 - A, B then C, D
 - C, D then A, B
 - ***Final balance will be \$3400***



Explanation (II)

- Account will not be credited if both functions read the old value of the balance and the *debit* function updates the file last
 - Disk accesses are done in the order
 - A, C then B, D
 - C, A then B, D
 - ***Final balance will be \$400***



Explanation (III)

- Account will not be debited if both functions read the old value of the balance and the ***credit*** function updates the file ***last***
 - Disk accesses are done in the order
 - A, C then D, B
 - C, A then D, B
 - ***Final balance will be \$3500***



The problem

- We have a ***race condition***
 - Outcome of concurrent execution of the processes will depend on process speeds
- We will observe ***irreproducible errors***
 - ***Hard to detect and debug***



The solution

- Define ***critical sections*** in the two functions

- Control access to critical sections



The credit function

```
do_credit(int account, float amount) {  
    float balance;  
    fp = fopen(...); // open file  
    fseek(fp,...); // locate account  
    fscanf(fp,"%f", &balance); // read balance  
    balance += amount; // update balance  
    fseek(fp,...); // go back to account  
    fprintf(fp,"%f", balance); // save result  
} // do_credit
```



The debit function

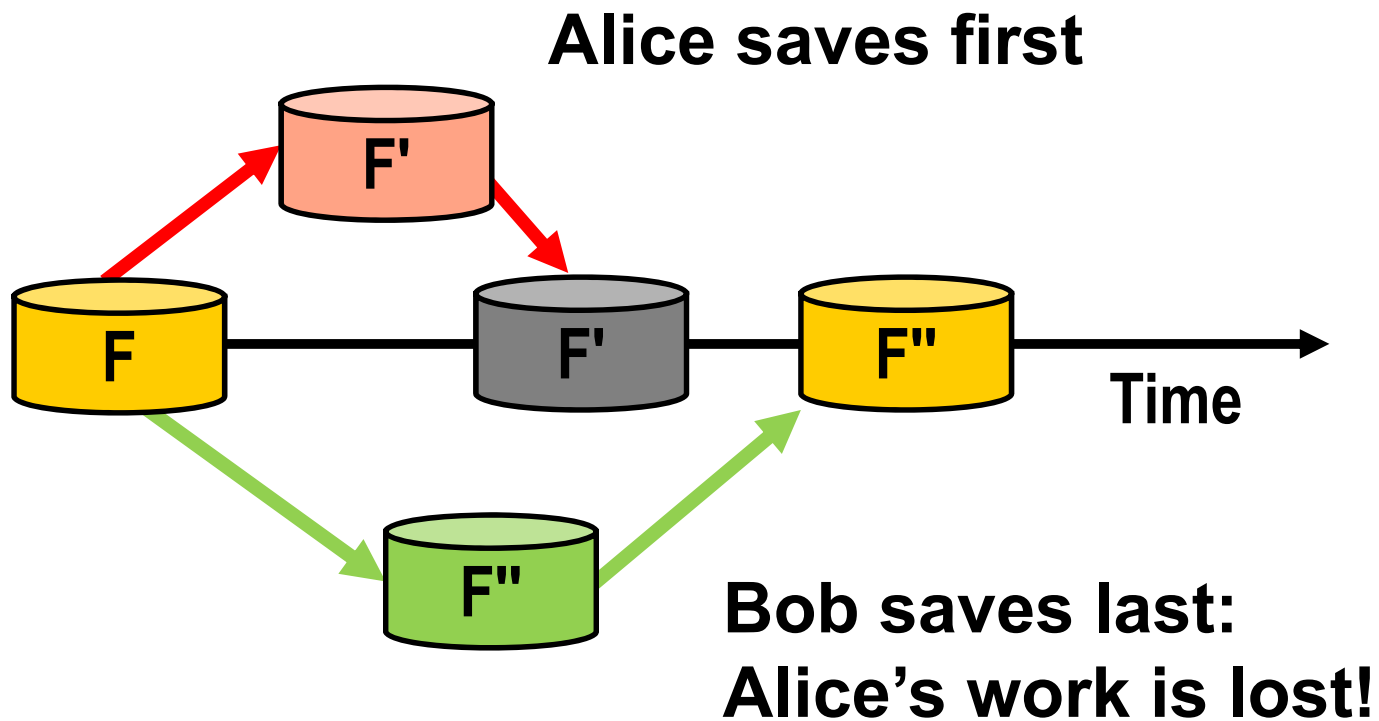
```
do_debit(int account, float amount) {  
    float balance;  
    fp = fopen(...); // open file  
    fseek(fp,...); // locate account  
    fscanf(fp, "%f", &balance); // read balance  
    balance -= amount; // update balance  
    fseek(fp,...); // go back to account  
    fprintf(fp, "%f", balance); // save result  
} // do_debit
```




Second example (I)

- Most text editors and word processors do not update directly the files on which they operate
 - Create a temporary *tempfile*
 - Modify the original file when the user issues a **save** command
- ***What will happen if two users edit the same file at the same time?***

Second example (II)





The problem

- Another ***race condition***
 - Should never have two users/processes updating the same file at the same time
- *Don't let other people edit our files*
 - *Send them a copy!*



A real race condition

- Pthread program creating a child thread
- Set a global counter to 0
- Both parent and child add one to counter 100,000 times
- When both are done, final counter value is 200,000
 - Right?



Realrace.cpp (I)

- `#include <iostream>`
`#include <pthread.h>`
`using namespace std;`
- `//Shared counter`
`static int counter = 0;`



Realrace.cpp (II)

- `// The thread function`
`void *child_fn(void *arg) {`
 `int i, ntimes;`
 `ntimes = (int) arg;`
 `for (i = 0; i < ntimes ; i++) {`
 `counter++;`
 `} // for`
`} // child_thread`



Realrace.cpp (III)

```
■ void main() {  
    pthread_t tid;  
    int ntimes = 100000;  
    int i = 0;  
  
    pthread_create(&tid, NULL, child_fn,  
                  (void *) ntimes);
```



Realrace.cpp (IV)

- ```
for (i = 0; i < ntimes ; i++) {
 counter++;
} // for
pthread_join(tid, NULL);
cout << "Final value of counter: " << counter
 << endl ;
} // main
```






# The outcome

- Final value of counter is ***smaller*** than 200,000
- Frequent ***race conditions*** result in conflicting increments
  - Both parent and child access the same counter
  - Increment it ***twice***



# Searching for a solution

- Three approaches
  - Disabling interrupts
  - Enforcing mutual exclusion 
  - Using an atomic transaction mechanism



# Disabling interrupts

- ***Very dangerous*** solution for user programs
- Was sometimes used for ***very short*** critical sections in the kernel
- Does not work for multithreaded kernels running on multiprocessor architectures.



# Enforcing mutual exclusion

- Portions of code where the shared data are read and modified are defined as **critical sections**
- **Must guarantee** that we will never have more than only one process in a critical section for the same shared data
- Works best with *short critical regions*
- *Preferred solution*



# Norace.cpp (I)

- `#include <iostream>`  
`#include <pthread.h>`  
`using namespace std;`
- `//Shared counter and shared lock`  
`static int counter = 0;`  
`static pthread_mutex_t lock;`

## Norace.cpp (II)

- // The thread function

```
void *child_fn(void *arg) {
 int i, ntimes;
 ntimes = (int) arg;
 for (i = 0; i < ntimes ; i++) {
 pthread_mutex_lock(&lock);
 counter++;
 pthread_mutex_unlock(&lock);
 } // for
} // child_thread
```



## Norace.cpp (III)

```
■ void main() {
 pthread_t tid;
 int ntimes =100000;
 int i = 0;
 // Create the lock
 pthread_mutex_init(&lock, NULL);
 // Create the thread
 pthread_create(&tid, NULL, child_fn,
 (void *) ntimes);
}
```

## Norace.cpp (IV)

```
■ for (i = 0; i < ntimes ; i++) {
 pthread_mutex_lock(&lock);
 counter++;
 pthread_mutex_unlock(&lock);
 } // for

 pthread_join(tid, NULL);
 cout << "Final value of counter: " << counter
 << endl;
} // main
```






# The outcome

- Program is much slower
- Gets the expected result



# Using atomic transactions

- Will allow several processes to access the same shared data without interfering with each other
- Preferred solution for database access since
  - Most databases are shared
  - Their critical sections can be very long as they often involve a large number of disk accesses



# Criteria to satisfy (I)

- Any solution to the critical section problem should satisfy the ***four following criteria***
  1. No two processes may be simultaneously into their critical sections for the same shared data  
*We want to enforce mutual exclusion*



## Criteria to satisfy (II)

2. No assumptions should be made about the speeds at which the processes will execute.

*The solution should be general:*

*the actual speeds at which processes complete are often impossible to predict because running processes can be interrupted at any time!*



## Criteria to satisfy (III)

3. No process should be prevented to enter its critical section when no other process is inside its own critical section for the same shared data

*Should not prevent access to the shared data when it is not necessary.*



## Criteria to satisfy (IV)

4. No process should have to wait forever to enter a critical section

*Solution should not cause starvation*



# My mnemonics

- Solution should provide
  1. Mutual exclusion
  2. All the mutual exclusion
  3. Nothing but mutual exclusion
  4. No starvation



# Solutions

- Five approaches
  - ***Pure software***
    - *Peterson's Algorithm*
  - ***Spin locks***
    - Use `xchg` instruction
  - ***Semaphores***
  - ***Monitors***
  - ***Locks and condition variables***





# Pure Software Solutions

- Make no assumption about the hardware
  - *Peterson's Algorithm*
  - Easier to understand than *Dekker's algorithm*
    - Original solution



## A bad solution (I)

```
#define LOCKED 1
#define UNLOCKED 0
int lock = UNLOCKED; // shared
// Start busy wait
while (lock == LOCKED);
lock = LOCKED;
 Critical_Section(...);
// Leave critical section
lock = UNLOCKED;
```



## A bad solution (II)

- Solution fails if two or more processes reach the critical section *in lockstep* when the critical section is UNLOCKED
  - Will both exit the busy wait
  - Will both set Lock variable to LOCKED
  - Will both enter the critical section at the same time
- ***Which condition does this solution violate?***

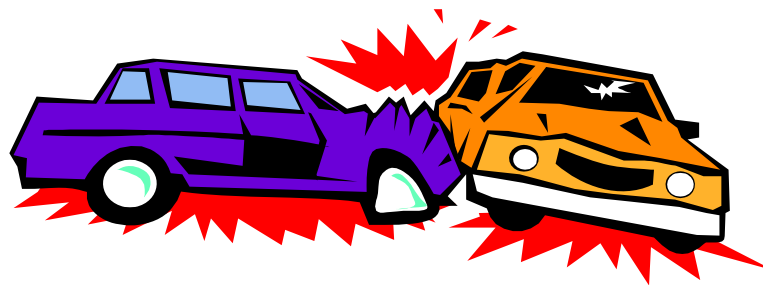


# Examples (I)

- You live in a dorm
  - Go to the showers
  - Find a free stall
  - Return to your room to bring your shampoo
  - Get into the free stall and find it occupied!

## Examples (II)

- You see a free parking spot
- Move towards it
- Do not see another car whose driver did not see you





## A bad solution (III)

- ***Solution violates second condition***
  - Does not ***always*** guarantee mutual exclusion



# Other bad solutions

- People have tried plenty of bad solutions

# Peterson Algorithm (I)

**NOT COVERED  
THIS SEMESTER**

- Simplest case
  - Two processes
  - Process IDs are **0** and **1**
- Philosophy is
  - Grab a lock for CS **before** checking that CS is free
  - Use a tie-breaker to handle conflicts



# Peterson Algorithm (II)

**NOT COVERED  
THIS SEMESTER**

```
#define F 0
#define T 1
// shared variables
int reserved[2] = {F, F};
int mustwait; // tiebreaker
```

# Peterson Algorithm (III)

**NOT COVERED  
THIS SEMESTER**

```
void enter_region(int pid) {
 int other; //other process
 other = 1 - pid;
 reserved[pid] = T;
 // set tiebreaker
 must_wait = pid;
 while (reserved[other]&&
 must_wait==pid);
} // enter_region
```



# Peterson Algorithm (IV)

**NOT COVERED  
THIS SEMESTER**

```
void leave_region(int pid) {
 reserved[pid] = F;
} // leave_region
```

# Peterson Algorithm (V)

NOT COVERED  
THIS SEMESTER

- Essential part of algorithm is

```
reserved[pid] = T;
must_wait = pid;
while reserved[other]&&
must_wait==pid);
```

- When two processes arrive *in lockstep*,  
*last one will wait*



# Spin locks



# Spin locks

- These solutions use special instructions to achieve an ***atomic test and set***:
  - Putting the lock testing and the lock setting functions into a single instruction makes the two steps ***atomic***
    - ***Cannot be separated by an interrupt***



# The exchange instruction (I)

- We assume the existence of a shared lock variable

```
int lockvar = 0; // shared
```

- Will only have two possible values
  - 0 meaning nobody is inside the critical section
  - 1 meaning somebody has entered the critical section



## The exchange instruction (II)

- We introduce an *atomic* instruction

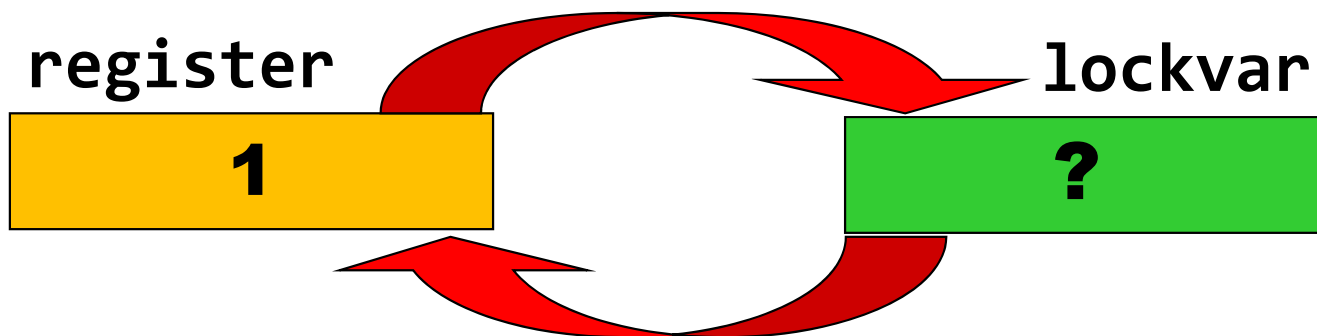
`exch register, lockvar`

- swaps contents of `register` and `lockvar`



# How we use it

- We set `register` to *one* before doing  
`exch register, lockvar`
- What will be the outcome?





# The two possible outcomes (I)

- If `register == 1`
  - `lockvar` was already set to 1
  - We **cannot enter** the critical section
  - We must **retry**
- If `register == 0`
  - `lockvar` was equal to 0
  - We have successfully set it to 1
  - We **have entered** the critical section

# Before the exchange

Register



Lockvar



- We do not know the state of lockvar
- Could be
  - Unlocked and free to grab?
  - Locked and in use by another process?

# After the exchange (I)

Register



Lockvar



- The lock was already locked
- Attempt failed
- Must retry

## After the exchange (II)

Register



Lockvar



- Lockvar was unlocked
- It is now locked
  - We succeeded!
- Can enter the critical region



# Entering a critical region

- To enter a critical region, repeat `exchange` until it succeeds

```
do {
 exchange(int &pv,int &plock)
 // pv points to register
 // plock points to lock var
} while (*pv == 1);
```



# Leaving a critical section

- To leave a critical region, do

```
*plockvar = 0;
```



# The x86 xchg instruction

- `xchg op1, op2`
  - Exchanges values of two operands
  - Always atomic (implicit **lock** prefix) if one of the operands is a memory address
    - `xchg %eax, lockvar`





# How to use it

- **enter\_region:**

```
movl 1, %eax # set to one
xchg %eax, lockvar
test %eax, %eax
jnz enter_region # try again
```
- **leave\_region:**

```
movl 0, %eax # reset to zero
xchg %eax, lockvar
```



## Same code in MASM

- `enter_region:`
  - `movl eax, 1 ; set to one`
  - `xchg eax, [lockvar]`
  - `test eax, eax`
  - `jnz enter_region ; try again`
- `leave_region:`
  - `movl eax, 0 ; reset to zero`
  - `xchg eax, [lockvar]`



# Underlying assumptions

- Peterson's algorithm and spinlocks assume that
  - Instructions execute in sequence
  - Instructions execute in an atomic fashion
- Less and less true in modern CPU architectures
  - Intel x86 architecture has an instruction prefix **lock** making any instruction writing into memory atomic
    - **lock movl 1, lockvar**



# The bad news

- Peterson's algorithm and spinlocks rely on ***busy waits***.
- Busy waits ***waste CPU cycles***:
  - Generate ***unnecessary context switches*** on single processor architectures
  - Slow down the progress of other processes



# Priority inversion

- A ***high priority process*** doing a busy wait may prevent a lower priority process to do its work and leave its critical region.
  - *Think about a difficult boss calling you every two or three minutes to ask you about the status of the report you are working on*



## In conclusion (I)

- **We *had to avoid* busy waits on single-core architectures**
- **We can use them only for *short waits on multicore architectures***



## In conclusion (II)

- Several operating systems for multiprocessor architectures offer two different mutual exclusion mechanisms:
  - ***Busy waits*** for very ***short waits***
    - ***Spinlocks***
  - Putting the waiting process in the ***blocked state*** until the resource becomes free for longer waits



## In conclusion (III)

- *Like waiting for a table in a restaurant:*
  - *If we are promised a very short wait, we will wait there*
  - *Otherwise, we might prefer to go for a walk (especially if it is a beach restaurant) or have a drink at the bar*





# Semaphores



# Semaphores

- Introduced in 1965 by E. Dijkstra
- Semaphores are special integer variables that can be initialized to any value  $\geq 0$  and can only be manipulated through two ***atomic operations***: **P( )** and **V( )**
- Also called **wait()** and **signal()**
  - Best to reserve these two names for operations on ***conditions*** in ***monitors***.



# The P( ) operation

- If semaphore value is zero,
  - Wait until value become positive
- Once value of semaphore is greater than zero,
  - Decrement it



# The V( ) operation

- Increment the value of the semaphore



# How they work

- The normal implementation of semaphores is through *system calls*:
  - ***Busy waits*** are ***eliminated***
  - Processes waiting for a semaphore whose value is zero are put in the ***blocked state***



# An analogy

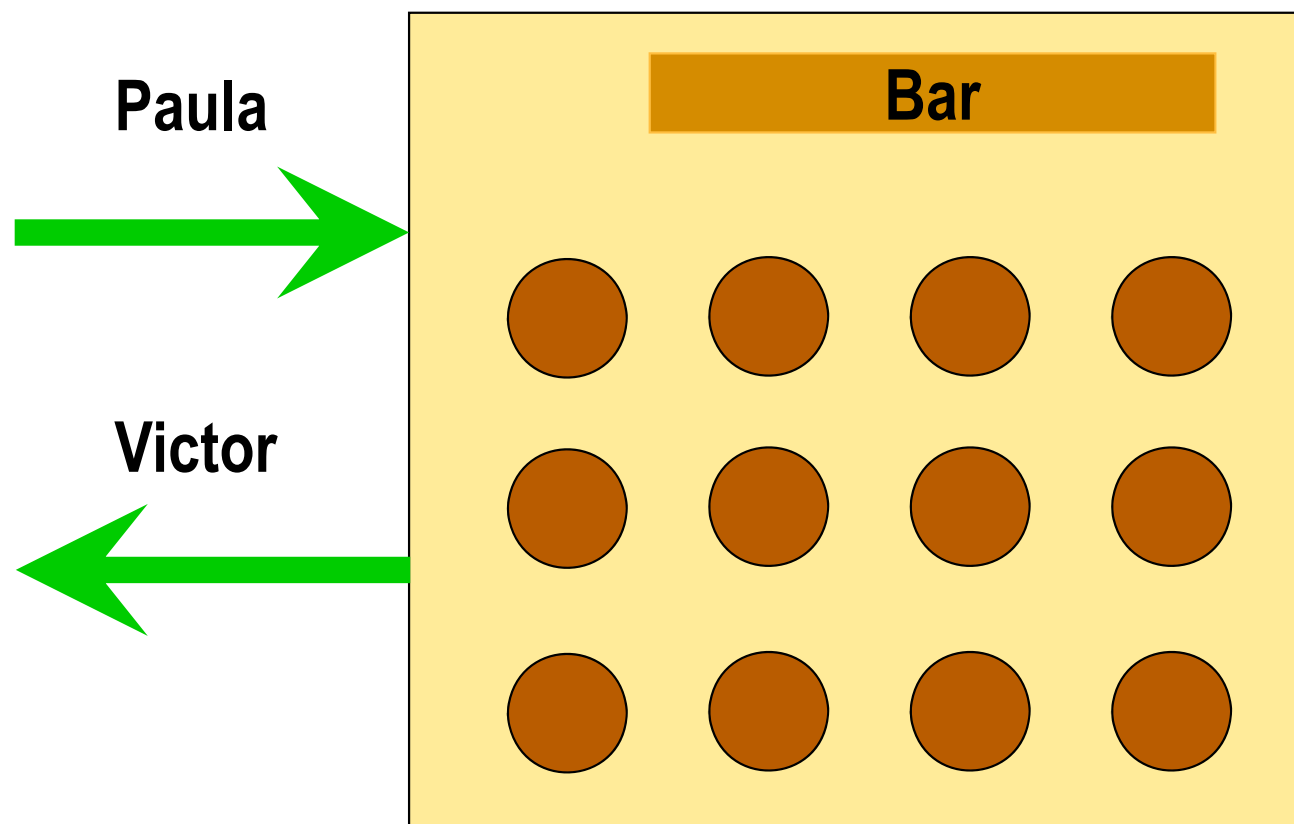
- Paula and Victor work in a restaurant:
- Paula handles customer arrivals:
  - Prevents people from entering the restaurant when all tables are busy.
- Victor handles departures
  - Notifies people waiting for a table when one becomes available



## An analogy (II)

- The semaphore represents the number of available tables
  - Initialized with the ***total number of tables*** in restaurant

## An analogy (III)



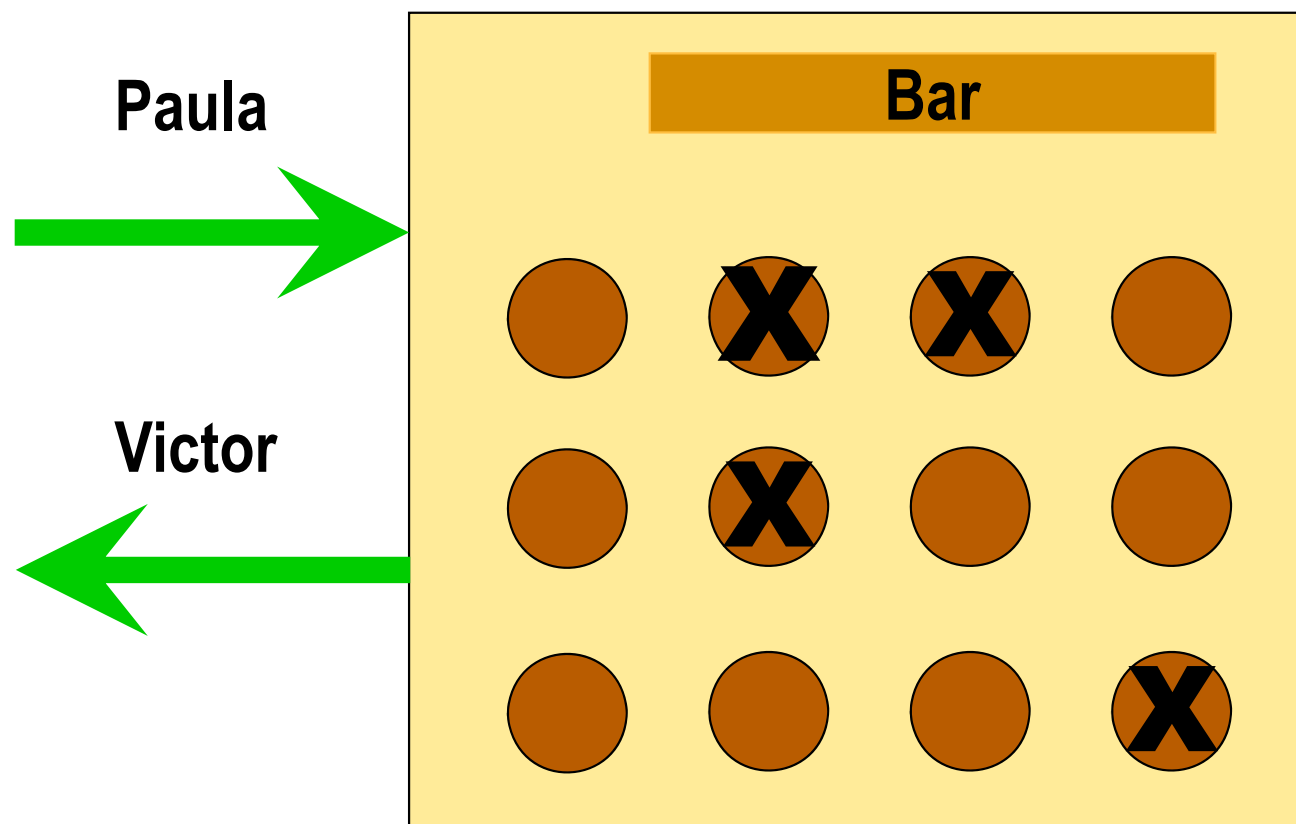




## An analogy (IV)

- When people come to the restaurant, they wait for Paula to direct them:
  - If a table is available, she let them in and decrements the table count
  - Otherwise, she directs them to the bar

# An analogy (V)





## An analogy (VI)

- When people leave, they tell Victor:
  - Victor increments the semaphore and checks the waiting area:
  - If there is anyone in there, he lets one group in and decrements the semaphore
- *Paula and Victor have worked long enough together and don't interfere with each other*



# Two problems

- What if somebody sneaks in the restaurant and bypasses Paula?
  - *Paula will let a group of people in when all tables are busy.*
- What if people forget to tell Victor they are leaving?
  - *Their table will never be reassigned.*



# Implementation (I)

- To avoid busy waits, we will implement semaphores as ***kernel objects***
- Each semaphore will have a ***value*** and an ***associated queue***.
- New system calls:
  - `sem_create( )`
  - `sem_P( )`:
  - `sem_V( )`
  - `sem_destroy( )`



## Implementation (II)

- `sem_create( )`:
  - Creates a semaphore and initializes it
- `sem_destroy( )`:
  - Destroys a semaphore



## Implementation (III)

- `sem_P( )`:
  - If the semaphore value is greater than zero, the kernel decrements it by one and lets the calling process continue.
  - Otherwise the kernel puts the calling process in the waiting state and stores its process-id in the semaphore queue.



# Implementation (IV)

- `sem_v( )`:
  - If the semaphore queue is not empty, the kernel selects one process from the queue and puts it in the ready queue
  - Otherwise, the kernel increments by one the semaphore value





# Binary semaphores

- Their value can only be **zero** or **one**
- Mostly used to provide **mutual exclusion**
- Semantics of P( ) operations not affected
- V( ) now sets semaphore value to one



# Mutual Exclusion (I)

- Assign one semaphore to each group of data that constitutes a critical section
- ***Initial value*** of semaphore must be ***one***:  
    **semaphore mutex = 1;**



## Mutual exclusion (II)

- Before entering a critical region, processes must do:
  - `P(&mutex);`
  - *Wait until critical region becomes free*
- Processes leaving a critical region must do
  - `V(&mutex);`
  - *Signal the process is leaving the critical section*



# Making a process wait

- The *initial value* of semaphore must be **zero**.  
`semaphore waitforme = 0;`
- Process that needs to wait for another process does:  
`sem_P(&waitforme);`
- When the other process is ready, it will do:  
`sem_V(&waitforme);`



# Example (I)

- Alice has promised to take her friends to the beach in her new car
- Everybody will meet on campus in front of the University Center
- Her three friends are Beth, Carol and Donna



## Example (II)

- We will have three semaphores

```
semaphore Beth_is_there = 0;
semaphore Carol_is_there = 0;
semaphore Donna_is_there = 0;
```

- There are all initialized to **zero**



## Example (III)

- Alice will do

```
sem_P(&Beth_is_there);
sem_P(&Carol_is_there);
sem_P(&Donna_is_there);
```



## Example (IV)

- When her friends arrive, they will do

```
sem_V(&Beth_is_there);
sem_V(&Carol_is_there);
sem_V(&Donna_is_there);
```





## Example (V)

- Our solution assumes that Alice will definitively be the first to arrive
  - *Her friends will never have to wait for her*
- If this is not the case, we need to force everyone to wait for everyone



# Setting up a rendezvous (I)

- To force two processes to wait for each other, we need **two** semaphores both initialized at **zero**

```
semaphore waitforfirst = 0;
```

```
semaphore waitforsecond = 0;
```



## Setting up a rendezvous (II)

- When the first process is ready, it will do

```
sem_V(&waitforfirst);
sem_P(&waitforsecond);
```
- When the second process is ready, it will do

```
sem_V(&waitforsecond);
sem_P(&waitforfirst);
```



## Setting up a rendezvous (III)

- What will happen if the first process does

```
sem_P(&waitforsecond);
sem_V(&waitforfirst);
```

and the second process does

```
sem_P(&waitforfirst);
sem_V(&waitforsecond);
```



## Setting up a rendezvous (IV)

- We will have a **deadlock**



# Advantages of semaphores (I)

- Semaphores are machine-independent
- They are simple but very general
- They work with any number of processes
- We can have as many critical regions as we want by assigning a different semaphore to each critical region



## Advantages of semaphores (II)

- We can use semaphores for synchronizing processes in an arbitrary fashion
- The key idea is ***layering***:
  - Pick a powerful and flexible mechanism that apply to many problems
  - Build later better user interfaces



# Implementations

- UNIX has three noteworthy implementations of semaphores:
  - The old System V semaphores
    - Now obsolete
  - The newer POSIX semaphores
    - ***For reference only***
  - The Pthread semaphores
    - ***For reference only***





# Overview

- Six operations:
  - `sem_open()`
  - `sem_wait()`
  - `sem_post()`
  - `sem_getvalue()`
  - `sem_close()`
  - `sem_unlink()`

**Will focus here  
on named POSIX  
semaphores**



# Sem\_open()

- Sole non-trivial call
  - Works like `open()` with `O_CREAT` option
    - Accesses the named semaphore
    - Creates it **if and only if** it did not already exist

**WARNING:** If you are debugging a program that has crashed, `sem_open` will not reinitialize any semaphore that has survived the crash



# Sem\_open syntax

```
■ sem_t *mysem;
char name[] = "Sem Name";
unsigned int initial_value;
mysem = sem_open(name,
 O_CREAT, 0600,
 initial_value);
```

**0600 prevents other users from  
accessing the new semaphore**



# Semaphore names

- Semaphores appear in the file system in subdirectory of `/dev/shm`
  - Names prefixed with "sem."
- Can be removed just like regular files using "rm"
- The names of the semaphores you are using must be ***unique***
  - All stored in a system wide directory



# A source of troubles

- `sem_open(...)` does ***not change*** the value of an existing semaphore
  - `initial_value` is only used if the semaphore did not already exist
- Must be sure that all your semaphores have been deleted before restarting your program
  - `ls /dev/shm/sem.*`



## Sem\_wait() and sem\_post()

- `sem_t *mysem;`  
`sem_wait(mysem);`
  - Implements the sem\_P() operation
- `sem_t *mysem;`  
`sem_post(mysem);`
  - Implements the sem\_V() operation



# Sem\_getvalue()

- Can test at any time the value of any opened semaphore:

```
sem_t *mysem;
int value;
sem_getvalue(mysem,&value);
```

- Non-standard feature of POSIX semaphores



# Sem\_close()

- `sem_t *mysem;`  
`sem_close(mysem);`
  - Closes the semaphore  
(without changing its value)





# Sem\_unlink()

- `char name[];`  
`sem_unlink(name);`
  - Removes the semaphore unless it is accessed by another process
    - That process will still be able to access the semaphore until it closes it



# Pthread synchronization

- Pthreads offer three synchronization primitives
  - ***POSIX semaphores***
    - Can use private semaphores
      - Unnamed
  - ***Mutexes***
  - ***Condition variables***

**For reference only**



# Unnamed semaphores (I)

- Have no system-wide name in `/dev/shm`
- Work like regular POSIX semaphores but for creation and deletion
  - `sem_init()` and `sem_destroy()`
- Given a regular variable name
- To remain visible to all users, variable must be declared
  - Static if only shared by pthreads
  - In a shared memory segment if shared by processes



## Unnamed semaphores (II)

- Creating a semaphore

- `static sem_t sem;`  
`int sem_init(sem_t *sem, int pshared,  
              unsigned initial_value);`

- If `pshared` is 0

- Semaphore is only shared by the threads *within* the process
    - Otherwise it can be shared with *other* processes



# Mutexes

- Built-in mutexes
  - `static pthread_mutex_t name;`
  - `pthread_mutex_init(&name, NULL);`
  - `pthread_mutex_lock(&name);`
  - `pthread_mutex_unlock(&name);`
- Mutexes are always initialized to ***one***
  - As they should



# Mutexes and binary semaphores

## ***Binary semaphores***

- Can only have two values
- Any process can lock or unlock a binary semaphore
  - Can be used for rendezvous

## ***Mutexes***

- Can only have two values
- ***A mutex can only be unlocked by the thread that locked it***
  - Less general



# Classical synchronization problems



# What are they?

- Will cover three problems
  - Bounded buffer
  - Readers and writers
  - Dining philosophers
- Will mention but not cover
  - Sleeping barber

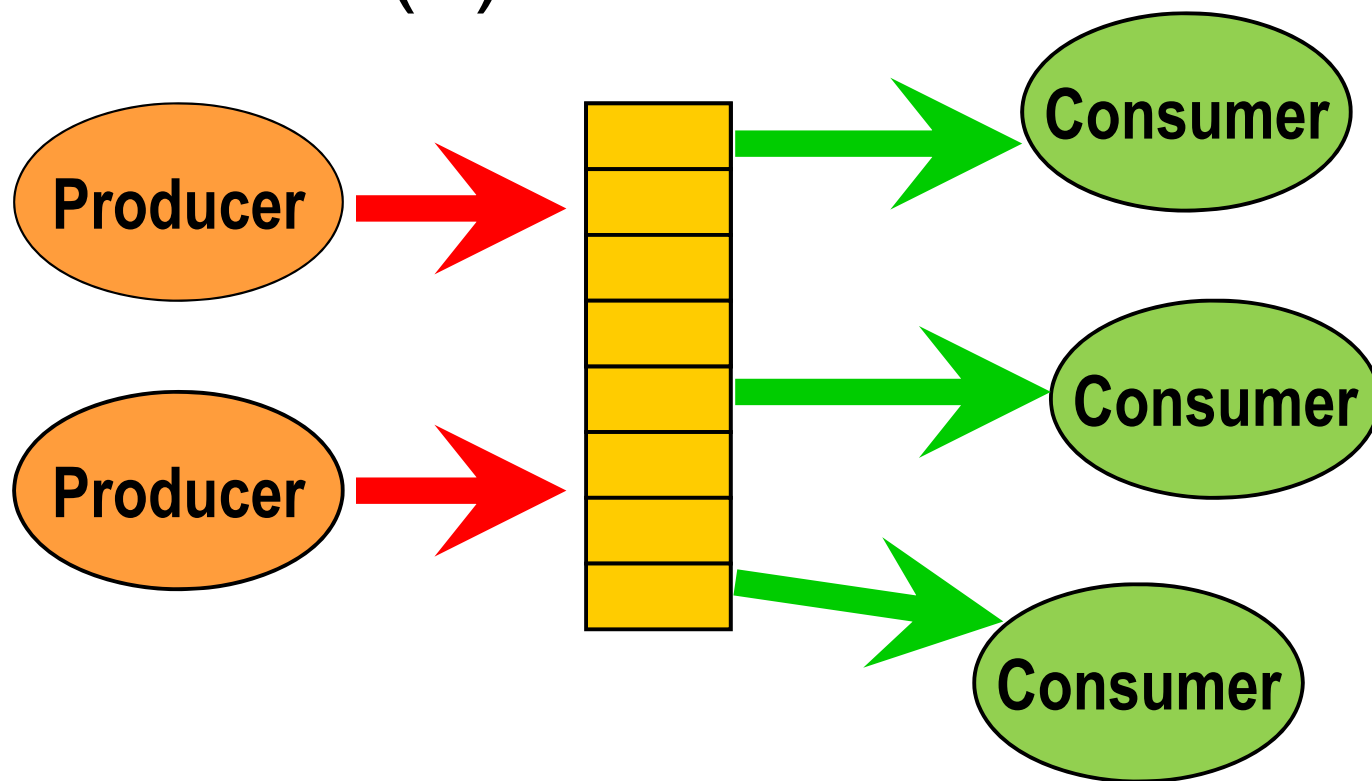




# Bounded buffer (I)

- One or more producer processes put their output in a bounded buffer
  - Must wait when buffer is full
- One or more consumer processes take items from the buffer
  - Must wait when buffer is empty

## Bounded buffer (II)





# The three rules

- Producers cannot put items in the buffer when it is full
- Consumers cannot take items from the buffer when it is empty
- Producers and consumers must access the buffer one at a time



# Two analogies

## ■ **The supermarket**

- Supermarket is the buffer
- We are the consumers
- Suppliers are the producers

## ■ **Our garbage**

- Our garbage can is the buffer
- We are the producers
- Garbage truck is the consumer



# The solution

- Declarations

```
#define NSLOTS ... // size
semaphore mutex = 1;
semaphore notfull = NSLOTS;
semaphore notempty = 0;
```

# The functions

```
■ producer() {
 struct x item;
 for (;;) {
 produce(&item);
 sem_P(¬full);
 sem_P(&mutex);
 put(item);
 sem_V(&mutex);
 sem_V(¬empty);
 } // for
} // producer
```

```
consumer() {
 struct x item;
 for (;;) {
 sem_P(¬empty);
 sem_P(&mutex);
 take(item);
 sem_V(&mutex);
 sem_V(¬full);
 consume(item);
 } // for
} // consumer
```

# A bad solution

```
■ producer() {
 struct x item;
 for (;;) {
 produce(&item);
 sem_P(¬full);
 sem_P(&mutex);
 put(item);
 sem_V(&mutex);
 sem_V(¬empty);
 } // for
} // producer
```

```
consumer() {
 struct x item;
 for (;;) {
 sem_P(&mutex);
 sem_P(¬empty);
 take(item);
 sem_V(&mutex);
 sem_V(¬full);
 consume(item);
 } // for
} // consumer
```



# Order matters

- The order of the two **P( )** operations is ***very important***
  - Neither the producer or the consumer should request exclusive access to the buffer before being sure they can perform the operation they have to perform
- The order of the two **V( )** operations does not matter





# The readers-writers problem (I)

- We have a file (or a database) and two types of processes:
  - **Readers** that need to access the file
  - **Writers** that need to update it.
- *A real problem*



## The readers-writers problem (II)

- Readers must be prevented from accessing the file while a writer updates it.
- Writers must be prevented from accessing the file while any other process accesses it
  - They require *mutual exclusion*



# An analogy

- Sharing a classroom between teachers and students
  - Teachers use it to lecture
    - *They cannot share the room*
  - Students use it for quiet study
    - *They can share the room with other students*
- Classroom is assumed to be in use if the light is turned on



# Rules for teachers

- Do not enter a classroom if its light is turned on
- Otherwise
  - Turn the light on when you come in
  - Turn the light off when you leave



# Rules for students

- If the light is on and you see students but no teacher
  - Enter the room
- If the light is off, you are the ***first student*** to enter the room
  - Turn the light on and enter the room
- If you are the ***last student*** to leave the room
  - Turn the light off after leaving



# The readers-writers problem (III)

- Shared variables and semaphores

```
int readers_count = 0;
semaphore mutex = 1;
semaphore access = 1;
```



## The readers-writers problem (IV)

```
write_to_file() {
 P(&access);
 . . .
 V(&access);
} // write_to_file
```

# The readers-writers problem (V)

```
read_the_file(){
 readers_count++;
 if(readers_count == 1)
 sem_P(&access);
 . . .
 readers_count--;
 if(readers_count == 0)
 sem_V(&access);
} // read_the_file
```

**TENTATIVE  
SOLUTION**





# Classrooms with two doors

- What if ***two students enter in lockstep*** using different doors?
  - Second will think he is the first to enter the room
  - Will see the light on and not enter
- What if ***two students leave in lockstep*** using different doors?
  - Neither of them will notice they are the last ones to leave
  - Neither will turn the light off



## The readers-writers problem (VI)

```
read_the_file(){
 sem_P(&mutex);
 readers_count++;
 if(readers_count == 1)
 sem_P(&access);
 sem_V(&mutex);
 . . .
```



## The readers-writers problem (VII)

```
sem_P(&mutex);
readers_count--;
if(readers_count == 0)
 sem_V(&access);
sem_V(&mutex);
} // read_the_file
```



# Starvation

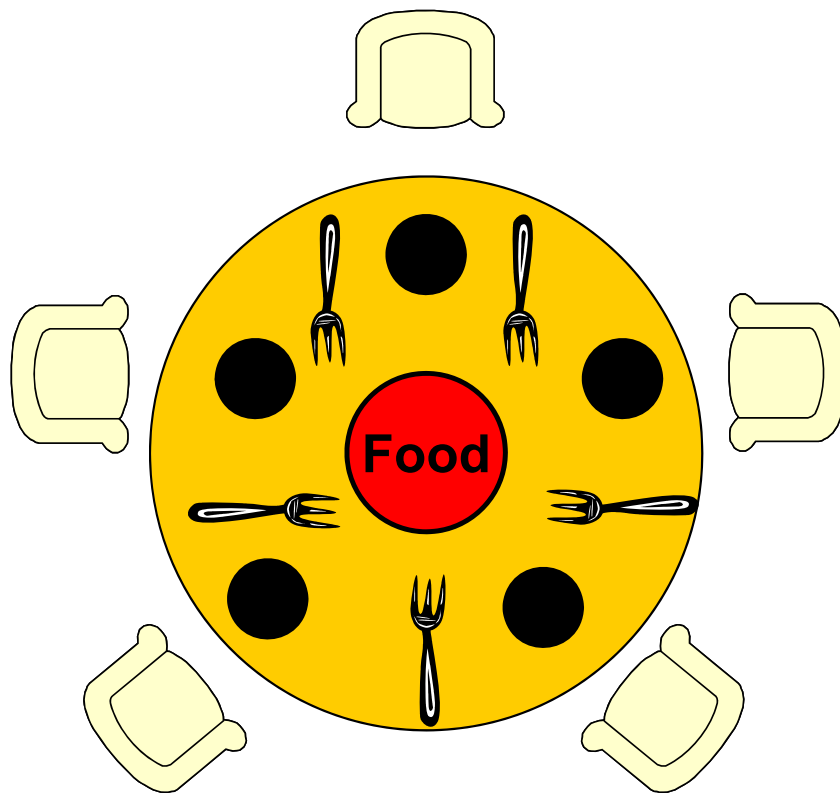
- Solution favors the readers over the writers
  - A continuous stream of incoming readers could block writers forever
    - ***Result would be writers' starvation.***



# The dining philosophers (I)

- Five philosophers sit at a table. They spend their time thinking about the world and eating spaghetti
  - The problem is that there are only five forks.
  - If all five philosophers pick their left forks at the same time, a ***deadlock*** will occur

# The table layout





# The dining philosophers (II)

```
#define N 5
semaphore mutex = 1;
```



## The dining philosophers (III)

```
philosopher(int i) {
 for (;;) {
 think();
 take_fork(i); // left before
 take_fork((i+1)%N); // right
 eat();
 put_fork(i);
 put_fork((i+1)%N);
 } // for loop
} // philosopher
```



# Avoiding the deadlock

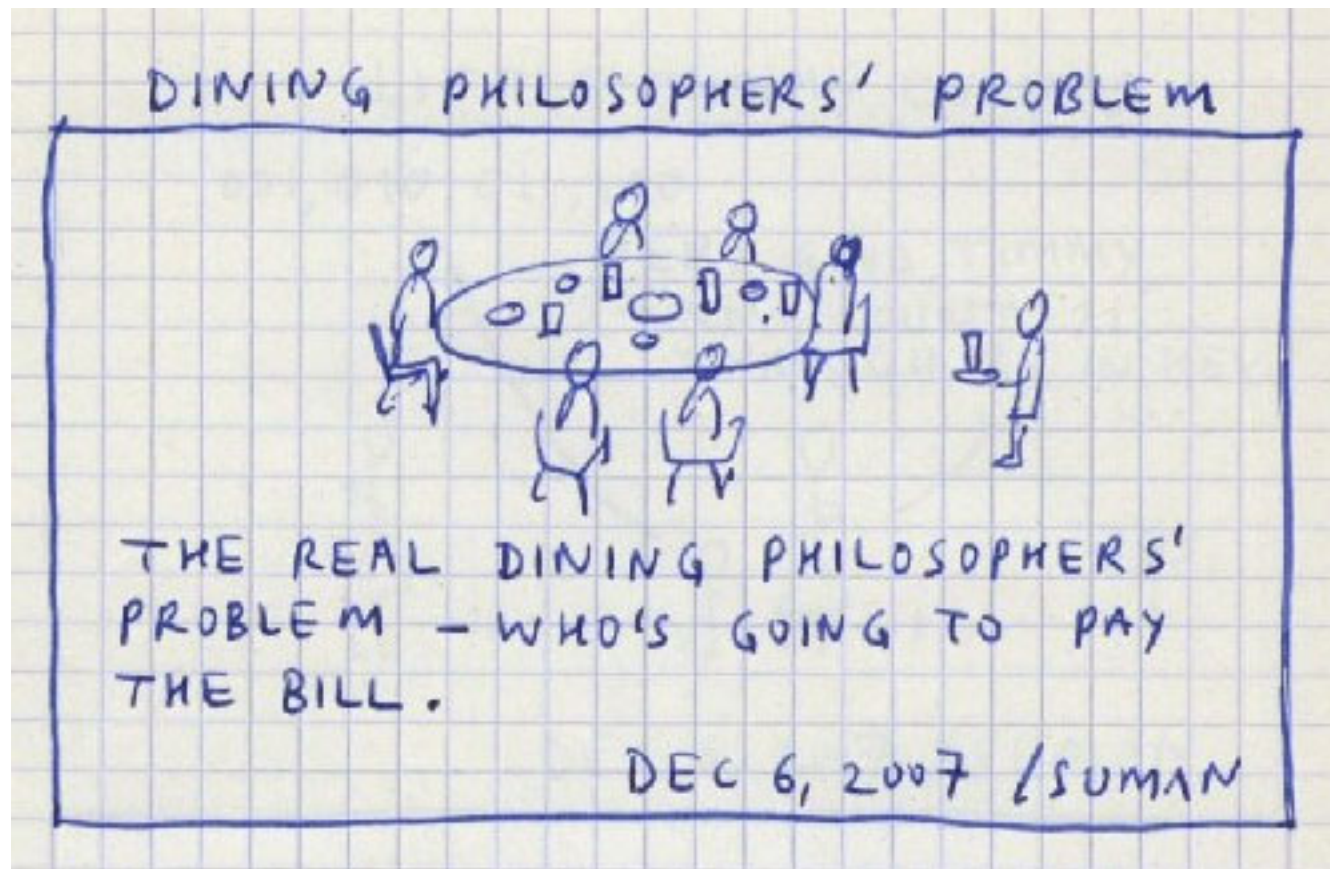
```
philosopher(int i) {
 for (;;) {
 think();
 if (i == 0) {
 take_fork((i+1)%N); // right
 take_fork(i); // before left
 } else {
 take_fork(i); // left before
 take_fork((i+1)%N); // right
 } // if-else
 ...
 } // for loop
} // philosopher
```



## The dining philosophers (IV)

- To break the deadlock, we force one of the philosophers to grab their *right fork before* their *left fork*
- *The main interest of this problem is that it belongs to the operating system folklore*

# The dining philosophers (V)





# The sleeping barber (I)

- Proposed by Andrew Tanenbaum in his textbook *Modern Operating Systems*.
  - ***Not covered it in class.***
  - Shows how to track the value of a semaphore using a global variable.



## The sleeping barber (II)

- A barber shop has several chairs for waiting customers and one barber who sleeps when there are no customers.
- Customers don't wait if the shop is full and there are no free chairs to sit upon.
  - Must keep track of the number of customers in the shop



# Global declarations

- ```
#define NCHAIRS 4
// number of chairs
semaphore mutex = 1; semaphore ready_barber = 0;
semaphore waiting_customers = 0;
int nwaiting = 0;
// tracks value of waiting_customers
```



The barber function

```
■ barber() {  
    for(;;) {  
        sem_P(&waiting_customers);  
        sem_P(&mutex);  
        nwaiting--;  
        sem_V(&ready_barber);  
        sem_V(&mutex);  
        cut_hair();  
    } // for  
} // barber
```



The customer function

```
■ customer() {  
    P(&mutex);  
    if (nwaiting < NCHAIRS) {  
        nwaiting++;  
        sem_V(&waiting_customers);  
        V(&mutex);  
        sem_P(&ready_barber);  
        get_haircut();  
    } // if  
    sem_V(&mutex);  
} // customer
```




Limitation of semaphores

- Semaphore are a **low level** construct:
 - Deadlocks will occur if **V()** calls are forgotten
 - Mutual exclusion is not guaranteed if **P()** calls are forgotten
- *Same situation as FORTRAN **if** and **goto** compared to more structured constructs*



A better solution

- We need a ***programming language construct*** that guarantees mutual exclusion
 - Will ***not trust processes*** accessing the critical region
- We can build it on the top of semaphores



Monitors



Monitors

- A ***programming language construct*** introduced by Hoare (1974) & Brinch-Hansen (1975)
- Finally implemented in Java
 - without *named conditions*
- A ***monitor*** is a package encapsulating procedures, variables and data structures.



Monitors

- To access the monitor variables, processes ***must*** go through one of the ***monitor procedures***.
- Monitor procedures are always executed ***one at a time***
 - Mutual exclusion is ***always guaranteed***.

User view





Monitors

- Monitor procedures can
 - ***wait*** on a ***condition*** (`cond.wait`)
 - until they get a ***signal*** (`cond.signal`) from another monitor procedure.
- Although conditions look like normal variables, they have ***no value***



Monitors

- *If a monitor procedure signals a condition and no other procedure is waiting for it, the signal will be lost:*
- ***It does not help to scream when nobody is listening!***



Monitors

- *If a monitor procedure waits for a condition that has already been signaled, it will remain blocked until the condition is signaled again :*
- ***It does not help either to wait for something that has already happened!***



Not the same as semaphores

- If a process does a V() operation on a semaphore and no other process is doing a P() operation on the semaphore, the value of the semaphore will be changed
- This is ***not true*** for condition variables



The monitor body

- The monitor **body** is executed when monitor is started
 - Its major purpose is to *initialize* the monitor variables and data structures.



First example (I)

- Implementing semaphores on top of monitors
 - Class semaphore with methods P() and V()
- No practical application
 - *Monitors are implemented on top of semaphores and not the other way around!*
- Shows that monitors are ***as powerful as semaphores***



First example (II)

```
Class semaphore {  
    // private declarations  
    private condition notzero;  
    private int value; // semaphore's value
```



First example (III)

```
// must be public and syn'd
public void synchronized sem_P(){
    // check before waiting
    if (value == 0)
        notzero.wait();
    value--; // decrement
} // P
```



First example (IV)

```
// must be public and syn'd  
public void synchronized sem_V(){  
    value++;  
    notzero.signal();  
} // V
```

Note that the V() method always signals the notzero condition even when it was already true



First example (V)

```
// constructor
semaphore(int initial_val){
    value = initial_val;
} //constructor
} // Class semaphore
```




Second example (I)

- The bounded buffer
 - Class `Bounded_Buffer` with methods `put()` and `get()`



Second example (II)

```
Class Bounded_Buffer {  
  
    // private declarations  
    private condition notfull;  
    private condition notempty;  
    private int bufsize;  
    private int nfullslots;
```



Second example (III)

```
// monitor procedures
// must be public and sync'd
public void synchronized put(){
    // MUST CHECK FIRST
    if (nfullslots == bufsize)
        notfull.wait();

    . . .
    nfullslots++;
    notempty.signal();
} // put
```



Second example (III)

```
// monitor procedures (cont'd)
// must be public and sync'd
public void synchronized get(){
    // MUST CHECK FIRST
    if (nfullslots == 0)
        notempty.wait();

    . . .
    nfullslots--;
    notfull.signal;
} // get
```



Second example (I)

- The bounded buffer
 - Class `Bounded_Buffer` with methods `put()` and `get()`



Second example (II)

```
Class Bounded_Buffer {  
  
    // private declarations  
    private condition notfull;  
    private condition notempty;  
    private int bufsize;  
    private int nfullslots;
```



Second example (III)

```
// monitor procedures
// must be public and sync'd
public void synchronized put(){
    // MUST CHECK FIRST
    if (nfullslots == bufsize)
        notfull.wait();

    . . .
    nfullslots++;
    notempty.signal();
} // put
```



Second example (IV)

```
// monitor procedures (cont'd)
// must be public and sync'd
public void synchronized get(){
    // MUST CHECK FIRST
    if (nfullslots == 0)
        notempty.wait();

    . . .
    nfullslots--;
    notfull.signal;
} // get
```




Second example (V)

```
// constructor is monitor body
Bounded_Buffer(int size) {
    nfullslots = 0;
    buffer_size= size;
} //constructor
```



Semantics of signal (I)

- Gives ***immediate control*** of the monitor to the procedure that was waiting for the signal
 - The procedure that issued the signal is then put ***temporarily on hold***
- Has ***no effect*** if there is no procedure waiting for the signal



Semantics of signal (II)

- Causes two types of problems
 - Too many context switches
 - Prevents programmers from putting signal calls inside their critical sections
 - Sole truly safe place to put them is at the ***end of procedure***
 - *Not an ideal solution as the programmer can forget to put them there*



The notify primitive (I)

- Introduced by Lampson and Redell in Mesa
 - Adopted by Gosling for Java
- When a monitor procedure issues a **condition.notify()**, the procedure that was waiting for the notify does not regain control of the monitor until the procedure that issued the signal
 - Terminates
 - Waits on a condition



The notify primitive (II)

- **Advantages:**

- Fewer context switches
- Programmers can put notify() calls anywhere



The notify primitive (II)

- **Very minor disadvantage:**

- Condition might not be true anymore

- Should replace *if* in

- `if(condition_is _false)`
`condition.wait()`


- By a *while*

- `while(condition_is _false)`
`condition.wait()`



Java implementation

- The Java equivalent of a monitor is a Java class whose access methods have been declared ***synchronized***
- Java does not support ***named conditions***:
 - When a synchronized method does a **`wait()`**, it cannot specify the condition it wants to wait on
- Java has ***notify()*** and ***notifyAll()***



Advanced pthread synchronization



The big idea

- Monitors are safer and easier to use than semaphores
 - But they are a language-based construct
- Want to provide the same ease of use through pthread functions
- Introduce condition variables

Back to the bounded buffer problem

```
■ producer() {  
    struct x item;  
    for (;;) {  
        produce(&item);  
        sem_P(&notfull);  
        sem_P(&mutex);  
        put(item);  
        sem_V(&mutex);  
        sem_V(&notempty);  
    } // for  
} // producer
```

```
consumer() {  
    struct x item;  
    for (;;) {  
        sem_P(&notempty);  
        sem_P(&mutex);  
        take(item);  
        sem_V(&mutex);  
        sem_V(&notfull);  
        consume(item);  
    } // for  
} // consumer
```



Condition variables

- Pthread feature
- Always used in conjunction with pthread mutexes
- Let threads to synchronize based upon the actual values of data
 - Buffer full/not full

The new approach

```
■ producer() {
    struct x item;
    for(;;) {
        produce(&item);
        mutex_lock(&buf);
        while (nfull == bsize)
            wait(&notfull, &buf);
        put(item);
        nfull++;
        signal(&notempty);
        mutex_unlock(&buf);
    } // for
} // producer
```

```
consumer() {
    struct x item;
    for(;;) {
        mutex_lock(&buf);
        while (nfull == 0)
            wait(&notempty, &buf);
        take(item);
        nfull--;
        signal(&notfull);
        mutex_unlock(&buf);
        consume(&item);
    } // for
} // consumer
```



Creating conditional variables

- Must be declared `pthread_cond_t`

- **Static method:**

```
pthread_cond_t mycv = PTHREAD_COND_INITIALIZER;
```

- **Dynamic method:**

```
pthread_cond_t mycv;
```

```
...
```

```
int pthread_cond_init(&cv, NULL);
```



Comments

- Using `pthread_cond_init()` lets the programmer set the optional `process-shared` attribute:
 - Allows the condition variable to be seen by threads in other processes.
 - Use `NULL` to specify the default
- We will not discuss
 - `pthread_condattr_init (attr)`
 - `pthread_condattr_destroy (attr)`



Deleting condition variables

- `int pthread_cond_destroy(&cv);`
 - As with `pthread_cond_init()`, `pthread_cond_destroy()` will return zero if successful and an error code otherwise.



Operations on condition variables

- Three operations:

- `pthread_cond_wait(&cv, &mutex)`

- `pthread_cond_signal(&cv)`

- `pthread_cond_broadcast(&cv)`



pthread_cond_wait ()

- **pthread_cond_wait(&cv, &mutex)**
 - Waits until condition variable **cv** is signaled
 - Mutex **mutex** must be **locked** and **owned** by calling thread
 - While waiting for the signal, calling thread releases **mutex**
 - Upon successful return, **mutex** will be **locked** and **owned** by the calling thread



Usage

```
■ pthread_mutex_lock(&buf);  
  while (nfull == 0)  
    pthread_cond_wait(&notempty, &buf);  
  . . .  
  pthread_mutex_unlock(&buf);
```

Could use an `if` but using a `while` is safer



Signaling a condition

- **pthread_cond_signal(&cv)**

- Unblocks at least one thread currently blocked on the condition variable **cv**

- **pthread_cond_broadcast(&cv)**

- Unblocks all threads currently blocked on the condition variable **cv**



Warning

- The thread calling `pthread_cond_broadcast()` or `pthread_cond_signal()` must own the mutex that the threads calling `pthread_cond_wait()` have associated with the condition variable during their waits
- Otherwise expect unpredictable behavior



Usage

- `pthread_mutex_lock(&buf);`
`while (nfull == 0)`
 `pthread_cond_wait(¬empty, &buf);`
 ...
`pthread_cond_signal(¬empty);`
`pthread_mutex_unlock(&buf);`

What it means

```
pthread_mutex_lock(&buf);  
...  
while (nfull == 0)  
    pthread_cond_wait(&notempty, &buf);  
...  
pthread_mutex_unlock(&buf);
```

```
pthread_mutex_lock(&buf);  
...  
pthread_cond_signal(&notempty);  
pthread_mutex_unlock(&buf);
```

The signal waking up a waiting thread must **own** the lock that the signal had released (and wants to regain)



Notes

- There is also a `pthread_cond_timedwait(...)`
 - Not covered
- For more details on pthreads, refer to the LLNL tutorial:
 - **POSIX Threads Programming**

<https://computing.llnl.gov/tutorials/pthreads/>