Threads and Synchronization

Thierry Sans

(recap) Processes

- A process is defined by its Process Control Block (PCB) that defines:
  - The execution state (running, waiting, ready)
  - The address space with code and data
  - The execution context (PC, SP, registers)
  - The resources (open files)
  - and so others ...

The cost of multi-processing

```
while (1) {
  int sock = accept();
  if ((child_pid = fork()) == 0) {
    // Handle client request
  } else {
    // Close socket
  }
}
```

Recall our Web Server example
we need to fork a child process for each request

- Create a new PCB
- Copy the address space and the resources
- Have the OS execute this child process (context switching)
- Use signals and pipes if the child wants to send information back to the parent process

A good but costly abstraction

✓ Good to avoid processes interfering with each other but ...

- Creating a process is costly (space and time)
- Context switching is costly (time)
- Inter-process communication is costly (time)

The need for cooperation

An application could have some sort of cooperating processes

- that all share the same code and data (address space)
- that all share the same resources (files, sockets, etc.)
- that all share the same privileges

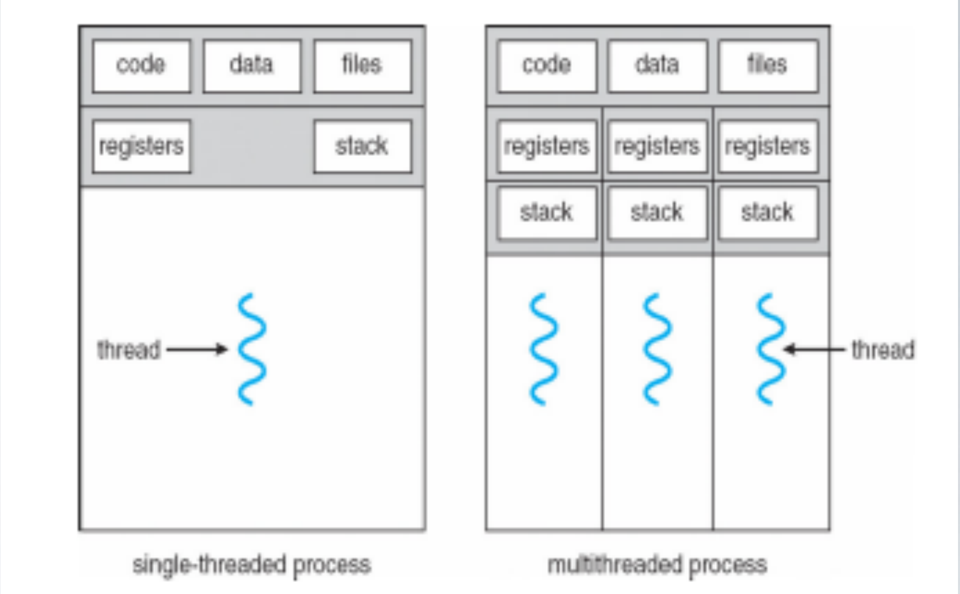
... while having different execution context (PC, SP, registers)

Rethinking process

Why not separate the process concept from its execution state?

- **Process** : address space, privileges, resources, etc
- **Thread** : PC, SP, registers

Threads



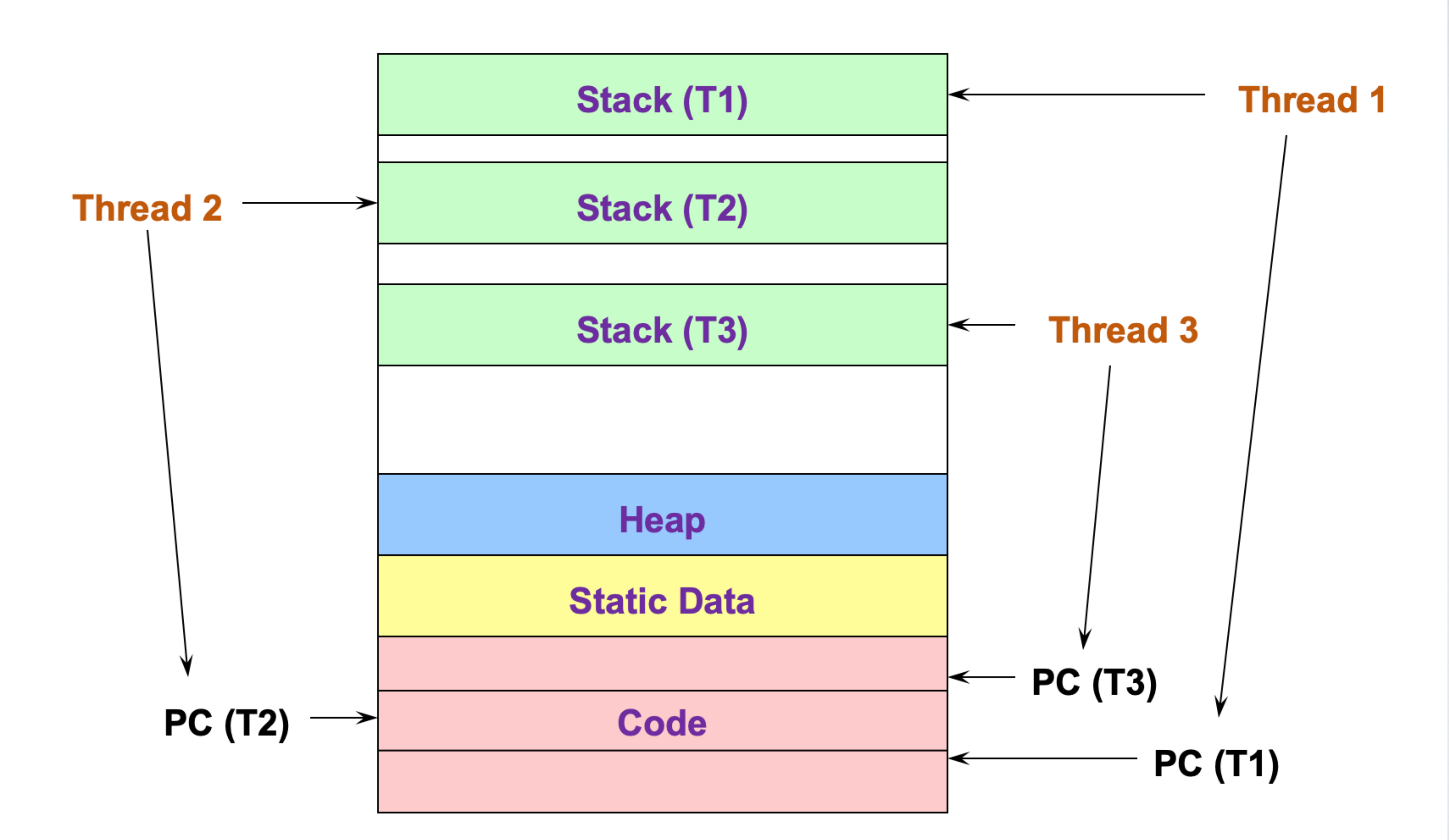
Modern OSes separate the concepts of processes and threads

- The thread defines a sequential execution stream within a process (PC, SP, registers)
- The process defines the address space and general process attributes (everything but threads of execution)

➡ Most popular abstraction for concurrency
threads become the unit of scheduling
while processes are now the containers in which threads execute

✓ A thread is bound to a single process but a process can have multiple threads

Threads within a process



Our web server becomes

```
web_server() {
    while (1) {
        int sock = accept();
        thread_fork(handle_request, sock);
    }
}

handle_request(int sock) {
    Process request
    close(sock);
}
```

Benefits

- **Responsiveness**
  an application can continue running while it waits for some events in the background

- **Resource sharing**
  threads can collaborate by reading and writing the same data in memory (instead of asking the OS to pass data around)

- **Economy of time and space**
  no need to create a new PCB and switch the entire context
  (only the registers and the stack)

- **Scalability in multi-processor architecture**
  the same application can run on multiple cores

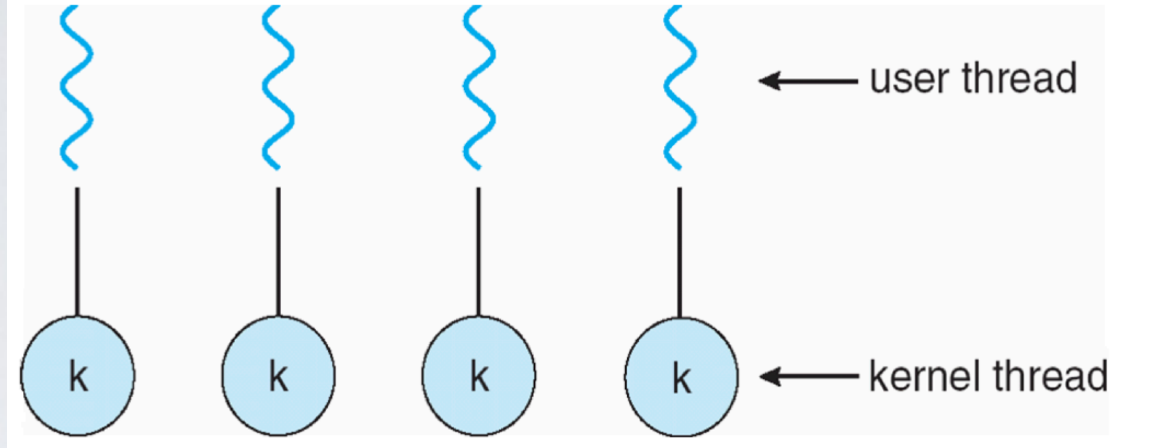
Multithreading models

- One-to-one model
  Kernel-level threads (a.k.a native threads)

- Many-to-one model
  User-level threads (a.k.a green threads)

- Many-to-many model
  Hybrid threads (a.k.a n:m threading)

One-to-one model
Kernel-level threads
(a.k.a native threads)



The kernel manage and schedule threads

- e.g Windows threads
- e.g POSIX pthreads `PTHREAD_SCOPE_SYSTEM`
- e.g (new) Solaris lightweight processes (LWP)

➡ All thread operations are managed by the kernel

✓ good for scheduling

✓ bad for speed

POSIX Thread API

- Create a new thread, run fn with arg

```
tid thread_create (void (*fn) (void *), void *);
```

  - Allocate Thread Control Block (TCB)
  - Allocate stack
  - Put func, args on stack
  - Put thread on ready list

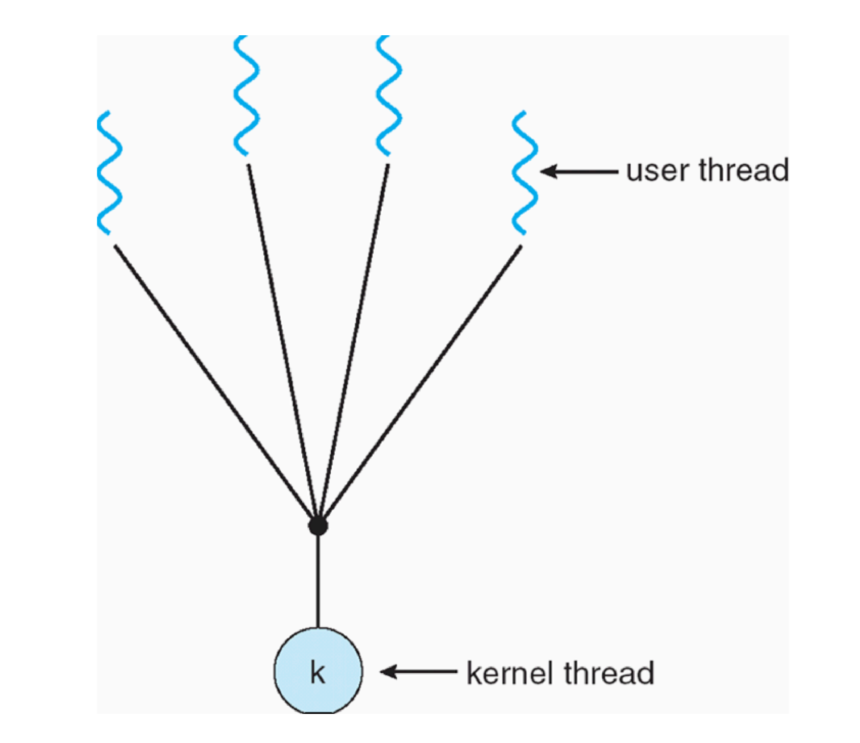
- Destroy current thread

```
void thread_exit ();
```

- Wait for thread thread to exit

```
void thread_join (tid thread);
```

Many-to-one model User-level threads (a.k.a green threads)



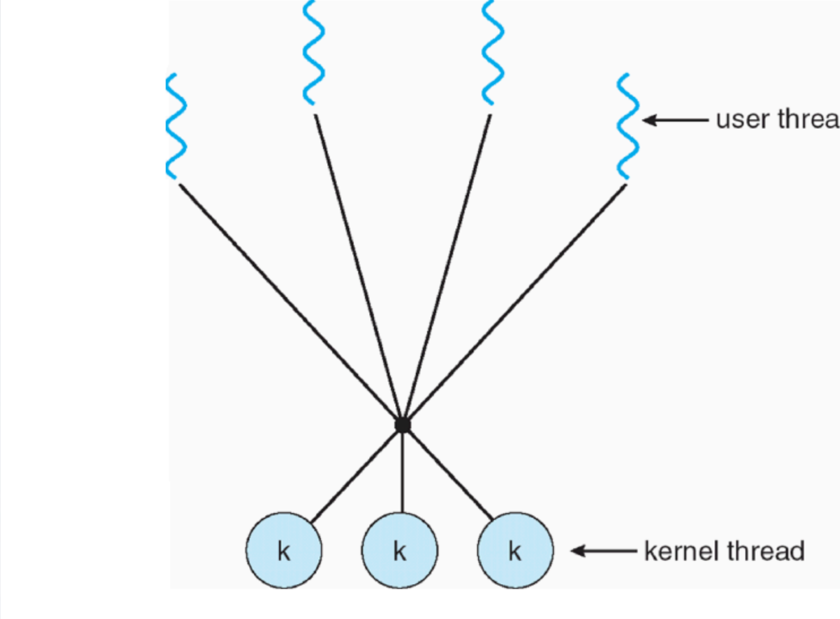
One kernel thread per process
thread management and scheduling is delegated to a library

- e.g pthreads `PTHREAD_SCOPE_PROCESS`
- e.g Java threads

➡ The kernel is not involved

✓ Very lightweight and fast
✓ All threads can be blocked if one of them is waiting or an event
✓ Cannot be scheduled on multiple cores

Many-to-many model Hybrid threads (a.k.a n:m threading)



User threads implemented on kernel threads

- e.g (old) Solaris

➡ Multiple kernel-level threads per process

Now threads can collaborate but ...

What are these two threads printing?

Ping thread

```
while(1){
  printf("ping\n");
};
```

Pong thread

```
while(1){
  printf("pong\n");
};
```

Too much milk

	Alice	Bob
12:30	Look in the fridge. Out of milk.	
12:35	Leave for store	
12:40	Arrive at store	Look in the fridge. Out of milk.
12:45	Buy milk	Leave for store
12:50	Arrive home, put milk away	Arrive at store
12:55		Buy milk
1:00		Arrive home, put milk away ... oh no!

Beyond milk

X is a global variable initialized to 0

thread 1

```
void foo(){
    x++;
};
```

thread 2

```
void bar(){
    x--;
};
```

What is the value of `x` after thread 1 and 2?

CPU instruction level

Incrementing (or decrementing) `x` is **not** an atomic operation

thread 1 (`foo` function)

```
LOAD X
INCR
STORE X
```

thread 2 (`bar` function)

```
LOAD X
DECR
STORE X
```

Non-deterministic execution

Execution scenario #1

```
LOAD X
INCR
STORE X
LOAD X
DECR
STORE X
```

➡ X is equal to 0

Execution scenario #2

```
LOAD X
LOAD X
INCR
DECR
STORE X
STORE X
```

➡ X is equal to −1

Execution scenario #3

```
LOAD X
LOAD X
INCR
DECR
STORE X
STORE X
```

➡ X is equal to 1

... and many other possible scenarios with the outcome of x being equal to either 0, −1 or 1

Race-condition problem

The system behaviours depends on the sequence or timing of events that is non-deterministic

- Not desirable in most cases (hard to catch bug)

Mutual exclusion

We want to use **mutual exclusion** to synchronize access to to shared resources

Code that uses mutual exclusion to synchronize its execution is called a **critical section**

- Only one thread at a time can execute in the critical section
- All other threads are forced to wait on entry
- When a thread leaves a critical section, another can enter

A classic example

Identify a critical section that lead to a race condition

```
Withdraw(acct, amt) {
   balance = get_balance(acct);
   balance = balance - amt;
   put_balance(acct, balance);
   return balance;
}
```

critical section

Requirements

1. **Mutual exclusion**
   If one thread is in the critical section, then no other is

➡ Mutual exclusion ensures ***safety property*** (nothing bad happen)

2. **Progress**
   If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section. A thread in the critical section will eventually leave it.

3. **Bounded waiting** (no starvation)
   If some thread T is waiting on the critical section, then T will eventually enter the critical section

➡ Progress and bounded waiting ensures the ***liveness property*** (something good happen)

4. **Performance**
   The overhead of entering and exiting the critical section is small with respect to the work being done within it

Mechanisms for building critical sections

- **Locks**
  Primitive, minimal semantics, used to build others
- **Semaphores**
  Basic, easy to get the hang of, but hard to program with
- **Monitors**
  High-level, requires language support, operations implicit

Locks

- A lock is an object in memory providing two operations
  - `acquire()`
    wait until lock is free, then take it to enter a C.S
  - `release()`
    release lock to leave a C.S, waking up anyone waiting for it

➡ Threads pair calls to `acquire` and `release`
We say that the thread *holds the lock* in between acquire/release

✓ Locks can spin (a spinlock) or block (a mutex)

Using locks

code

```
Withdraw(acct, amt) {
  acquire(lock);
  balance = get_balance(acct);
  balance = balance - amt;
  put_balance(acct, balance);
  release(lock);
  return balance;
}
```

execution of thread 1 and thread 2

```
acquire(lock);
balance = get_balance(account);
balance = balance - amount;
```

```
acquire(lock);
```

```
put_balance(acct, balance);
release(lock);
```

```
balance = get_balance(acct);
balance = balance - amt;
put_balance(acct, balance);
release(lock);
```

Implementing a spin lock (naive but wrong attempt)

```
struct lock {
    int held = 0;
}

void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}

void release (lock) {
    lock->held = 0;
}
```

What is the context switch happens in between?

➡ We have a race condition

The hardware to the rescue

- `test-and-set(TAS` x86 CPU instruction) atomically writes to the memory location and returns its old value in a **single indivisible step**

➡ the caller is responsible for testing if the operation has succeeded or not

```
bool test_and_set(bool *flag) {
  bool old = *flag;
  *flag = True;
  return old;
}
```

This is pseudo-code!
The hardware execute this atomically

Implementing a spin lock

```
struct lock {
    int held = 0;
}

void acquire (lock) {
    while test-and-set(&lock->held);
}

void release (lock) {
    lock->held = 0;
}
```

Busy wait (a.k.a spin)

- Waste of CPU time
- Unfair access to lock

Implementing a lock by disabling interrupt

```
struct lock {
}

void acquire (lock) {
    disable_interrupts();
}

void release (lock) {
    enable_interrupts();
}
```

- ➡ Disabling interrupts blocks notification of external events that could trigger a context switch
- ◉ Can miss or delay important events

Our lock implementations so far

- **Goal** : Use mutual exclusion to protect critical sections of code that access shared resources

✓ **Method** : Use locks (spinlocks or disable interrupts)

◉ **Problem** : Critical sections (CS) can be long

- ◉ spinlocks waste CPU time
  and do not provide fair access to lock
- ◉ disabling interrupts can delay important events

Blocking Lock Implementation

```
struct lock {
    int held = 0;
    queue Q;
}

void acquire (lock) {
     disable_interrupts();
     while (lock->held){
         enqueue(lock->Q, current_thread);
         thread_block(current_thread);
     }
     lock->held = 1;
     enable_interrupts();
}

void release (lock) {
    disable_interrupts();
    if (!isEmpty(lock->Q)){
        thread_unblock(dequeue(lock->Q));
    }
    lock->held = 0;
    enable_interrupts();
}
```

Semaphores

An abstract data type to provide mutual exclusion described by *Dijkstra* in the "*THE multiprogramming system*" in 1968

➡ Semaphores are "integers" that support two operations:

- `Semaphore::P()` decrement, block until semaphore is open
  a.k.a `wait()`, or `sem_wait()`, or `sema_down()`
- `Semaphore::V()` increment, allow another thread to enter
  a.k.a `signal()`, or `sem_post()`, or `sema_up()`

✓ Semaphore safety property
the semaphore value is always greater than or equal to 0

Blocking mechanism

Associated with each semaphore is a queue of waiting threads

➡ When `P()` is called by a thread:

- If semaphore is open, thread continue
- If semaphore is closed, thread blocks on queue

➡ Then `V()` opens the semaphore

- If a thread is waiting on the queue, the thread is unblocked
- If no threads are waiting on the queue, the signal is remembered for the next thread

Using semaphores

code

```
Withdraw(acct, amt) {
  P(s);
  balance = get_balance(acct);
  balance = balance - amt;
  put_balance(acct, balance);
  V(s);
  return balance;
}
```

execution of

thread 1, thread 2 and thread 3

```
P(s);
balance = get_balance(account);
balance = balance - amount;
```

```
P(s);
```

```
P(s);
```

```
put_balance(acct, balance);
V(s);
```

```
...
V(s);
```

```
...
V(s);
```

Semaphore Implementation

```
struct semaphore {
    int value;
    queue Q;
}

void init(sema, value){
    sema->value = value;
}

void P (sema) {
    disable_interrupts();
    while (sema->value == 0){
        enqueue(sema->Q, current_thread);
        thread_block(current_thread);
    }
    sema->value--;
    enable_interrupts();
}

void V (sema) {
    disable_interrupts();
    if (!isEmpty(sema->Q)){
        thread_unblock(dequeue(sema->Q));
    }
    sema->value++;
    enable_interrupts();
```

Two types of semaphores provided by OSes

Binary semaphore (a.k.a *mutex*)
controls access to a single resource (mutual exclusion)

➡ behave exactly like a lock

General semaphore (a.k.a counting semaphore)
controls access to a finite number of resources `n` available

- The semaphore is initialized with a positive integer n
- `P()` blocks when this number is equal to `0`

➡ at most `n` threads can be in the critical section

Acknowledgments

Some of the course materials and projects are from

- Ryan Huang - teaching CS 318 at *John Hopkins University*
- David Mazière - teaching CS 140 at *Stanford*