Operating Systems

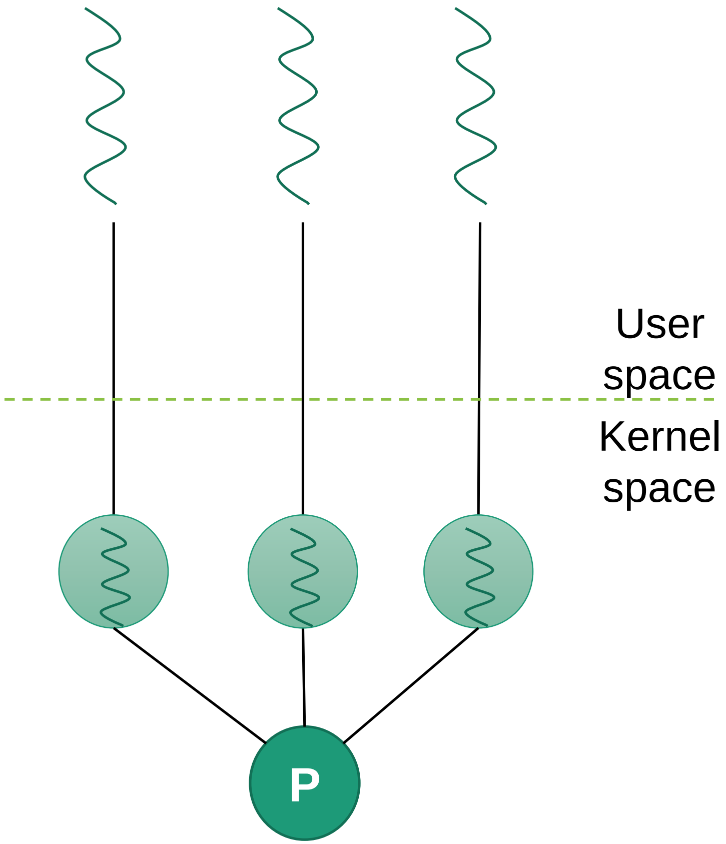
Recitation 8

Plan

- Thread synchronization
  - Lock & event in Linux
  - Mutex & Condition variable
  - What happens “under the hood”

Reminder

- Threads get scheduled by scheduler in kernel
- Preemptive multitasking:
  - OS decides when a thread will get its CPU time slot
- Context-switch without warning



Things can get ugly

```
void* mythread(void *arg) {
    char *letter = arg;
    int i;
    printf("%s: begin\n", letter);
    for (i = 0; i < TEN_MILLION; i++) {
        balance = balance + 1;
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

Code example

Things can get ugly

- `balance = balance + 1;`

- What really happens here? C translated to assembly
- In Intel x86* processor:

```
mov EAX, 0x8049a1c   // copy balance value
add EAX, 0x1         // increment by one
mov 0x8049a1c, EAX   // copy back
```

* http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Race condition

thread 1	thread 2	balance
`mov EAX, 0x8049a1c`		0
	`mov EAX, 0x8049a1c`	0
	`add EAX, 0x1`	0
	`mov 0x8049a1c, EAX`	1
`add EAX, 0x1`		1
`mov 0x8049a1c, EAX`		**1 (not 2!)**

- Result depends on the timing execution of the code
- Can get different result every time!

Let's take a closer look

- **`balance = balance + 1;`**

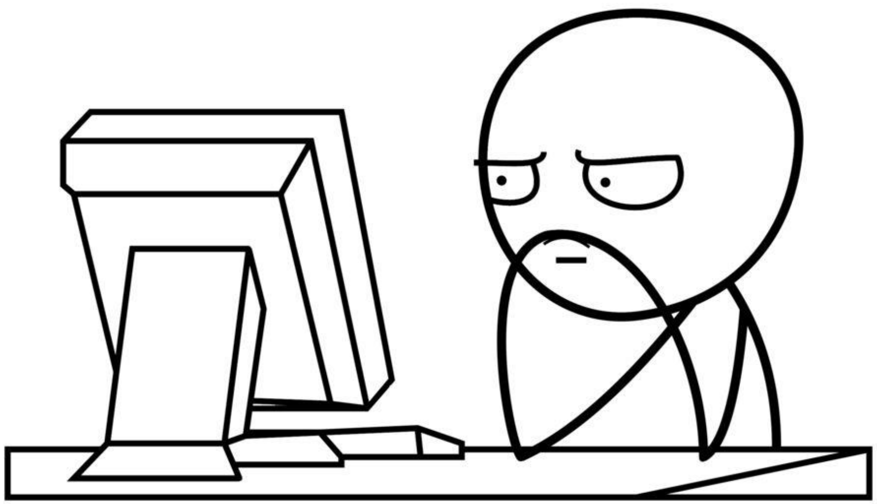
```
mov EAX, 0x8049a1c    // copy balance value
add EAX, 0x1          // increment by one
mov 0x8049a1c, EAX    // copy back
```

- What we have here is a **critical section**
- We need to access it with "atomicity", force it to be executed **<u>as a unit</u>**
  - guaranteed to finish without other threads running the same code (at same time)

Synchronization for dummies

- Crossing our fingers ☹
- Thread will run only after other thread ended ☹

- **Delays** ☹



Kernel sync mechanisms

- Atomic operations at **CPU level**!
  - Supported in instruction set
  - Example: intel x86 supports *atom_inc*, *atom_dec*, *atom_add*, *atom_sub*
- Interrupt disabling during critical section
  - No clock interrupt…
  - Hurts performance badly (unfair)
- Spinlocks (busy-wait)
  - Much more common
  - Saves context-switch
  - Useful only for very short periods

Linux Synchronization

- Main mechanism implemented in Linux for user-level synchronization
  - **Mutex** (lock)
  - **Condition variable** (signal)

- Windows
  - Critical section/Mutex
  - CreateEvent/WaitForSingleObject

Mutex

- Mutual exclusion
- Only one thread can hold it `lock()`ed
  - Others trying to `lock()` block until owner decides to free

- *Example:*

```
lock()
   //do some critical section code
   printf(“hello”)
unlock()
```

Mutex formal requirements

- Mutual exclusion
  - Only **one** thread can hold it `lock()`ed – can’t have two in same critical section
- Progress (deadlock freedom)
  - **Some** thread eventually enters critical section
- Starvation freedom
  - Thread wont starve, and will **eventually** enter critical section

It’s all in your head!

- Always remember when programming with mutexes:

  it’s a logical concept

- The protection of variables and code sections exists only *in your head*
- If you don’t consistently protect shared variables/critical code with a mutex, bad things will happen
- OS only provides the mechanism - you are the user!

Mutex API

```c
#include <pthread.h>
```

- Creation:

```c
int pthread_mutex_init(
      pthread_mutex_t *mutex,
      const pthread_mutex_attr_t *mutexattr);
```

- Destruction:

```c
int pthread_mutex_destroy(
      pthread_mutex_t *mutex);
```

Mutex API - Example

```
pthread_mutex_t lock;

int main() {
  if (pthread_mutex_init(&lock, NULL) != 0) {
    printf("mutex init failed\n");
    return 1;
  }
  /* … code here … */
  pthread_mutex_destroy(&lock);
}
```

Locking

```
int pthread_mutex_lock(
      pthread_mutex_t *mutex);
```

- If mutex is not free, block until it frees

```
int pthread_mutex_trylock(
      pthread_mutex_t *mutex);
```

- If mutex is not free, fail

```
int pthread_mutex_unlock(
      pthread_mutex_t *mutex);
```

- Free locked mutex

Under the hood

- Shared global variable acts as a ‘lock’
- Initially ‘unlocked’
  - `int mutex = 0;`
- Before entering critical section, a task ‘locks’ the mutex
  - `mutex = 1;`
- When done with critical section, ‘unlocks’ the mutex
  - `mutex = 0;`
- While mutex is “locked”, no other task can enter critical section

- **What’s the problem?**

Under the hood

- Special mutex variable needs to be accessed atomically
- Reasonable solution - hardware support
- One example (from the past):

  testandset *<address>*, rnew, rold

- Special <u>atomic</u> operation

```
int TestAndSet(int *lock, int new) {
    int old = *lock; // save old value of &lock in memory
    *lock = new;     // set new value
    return old;      // return old value
}
```

Simple implementation

```
void init() {
    // 0 means lock is available, 1 means held by a thread
    flag = 0;
}
void lock() {
    // busy-wait (do nothing)
    // exits loop only when old value is 0 == not locked!
    while (TestAndSet(&flag, 1) == 1) ;
}
void unlock() {
    flag = 0;
}
```

Simple implementation

```
void init() {
    // 0 means lock is available, 1 means held by a thread
    flag = 0;
}
void lock() {
    // busy-wait (do nothing)
    // exits loop only when old value is 0 == not locked!
    while (TestAndSet(&flag, 1) == 1) ;
}
void unlock() {
    flag = 0;
}
```

That's a LOT of ***spinning***! Too many time-slices wasted by scheduler on threads in hopeless loop

Also possibly ***starvation***! Doesn't ensure all threads will eventually acquire lock!

Less naive implementation

- Add `yield()` instruction

```
void init() {
    flag = 0;
}
void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up CPU on lock failure
}
void unlock() {
    flag = 0;
}
```

Not good enough

- Say we have 100 threads -
  - First threads locks, and gets preempted
  - 99 threads now try to `lock()`, fail and `yield()`
  - Still a LOT of context switching…
- And starvation…

More realistic implementation

- Implemented as struct with queue
  - Add thread to queue when lock unavailable
  - in `unlock()`, wake up one thread in queue

- A bit over-simplified
  - Also, mostly replaced by `Compare-and-Swap` (or other instructions)

Events

- Allow thread1 to inform thread2 on some event
  - Thread2 can sleep meanwhile
- Allow sync. access to sensitive shared resource
- Extension to mutex

Example: simple queue

- thread1 enqueues, thread2 dequeues
- Without sync. access:
  - Both threads may change data together
  - Thread1 insertion not safe (memory addresses…)
  - Thread2 won't know when to deq (memory addresses, polling…)

Condition Variables (1)

- Allow thread to sleep-wait() on event

```c
int pthread_cond_init(
      pthread_cond_t *cond,
      pthread_condattr_t *cond_attr);

int pthread_cond_destroy(pthread_cond_t *cond);
```

- Initialize/destroy condition variable object
  - cond_attr = NULL is default
- Destroy fails if threads are waiting

Condition Variables (2)

```
int pthread_cond_wait(
     pthread_cond_t *cond,
     pthread_mutex_t *mutex);
```

- Wait() on condition variable
- **Must have mutex already locked!**
- On success releases mutex and puts thread to sleep
- Several threads can wait()
  - But only one wakes up…

Condition Variables (3)

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- Signal a single wait()ing thread to wake up
- Choice of awakened thread is arbitrary

- Notice – **no mutex**

Back to queue example

```
item dequeue() {
    pthread_mutex_lock(&qlock);
    while <queue is empty>
        pthread_cond_wait(&notEmpty,&qlock);
    /* … remove item from queue … */
    pthread_mutex_unlock(&qlock);
    /* .. return removed item */
}
```

Why **while**?

Back to queue example

```
pthread_mutex_t qlock;
pthread_cond_t notEmpty;
/* … initialization code … */
void enqueue(item x) {
    pthread_mutex_lock(&qlock);
    /* … add x to queue … */
    pthread_cond_signal(&notEmpty);
    pthread_mutex_unlock(&qlock);
}
```

Another example: producer/consumer

- Thread 1 “produces” elements
  - Element counter
  - “Element to consume” variable

- Consumer threads “consume” elements
  - Wait on “Element to consume” variable
  - “consumes” it and notifies producer it’s ready for more

Code example