Operating Systems

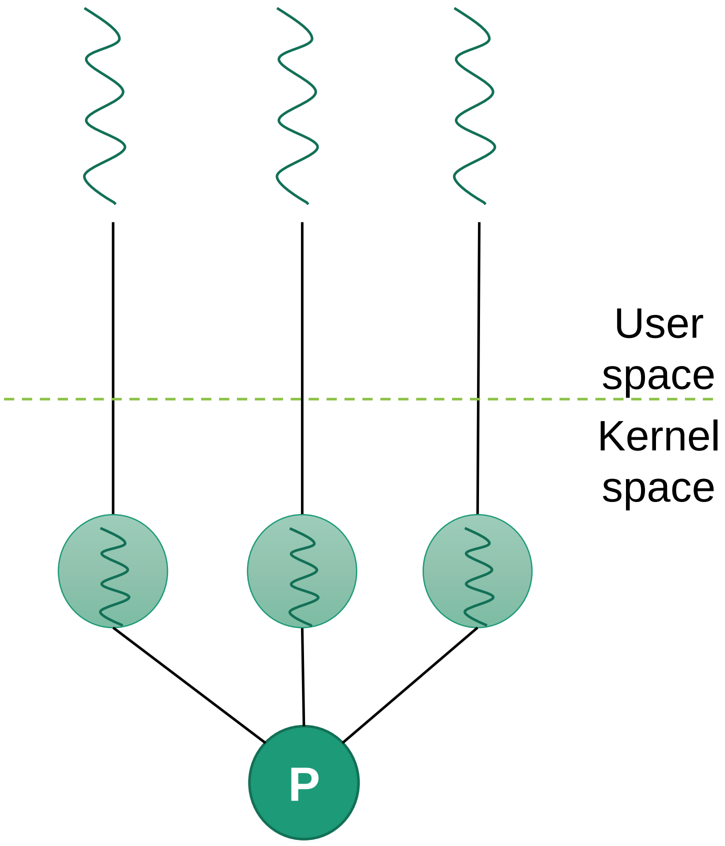
Recitation 8

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



Reminder

```
static int counter = 1;

int next_counter(void) {
  return counter++;
}
```

OK for single thread,
not for concurrent threads

What It Means

```
static int counter = 1;

int next_counter(void) {
  return counter++;
}
```

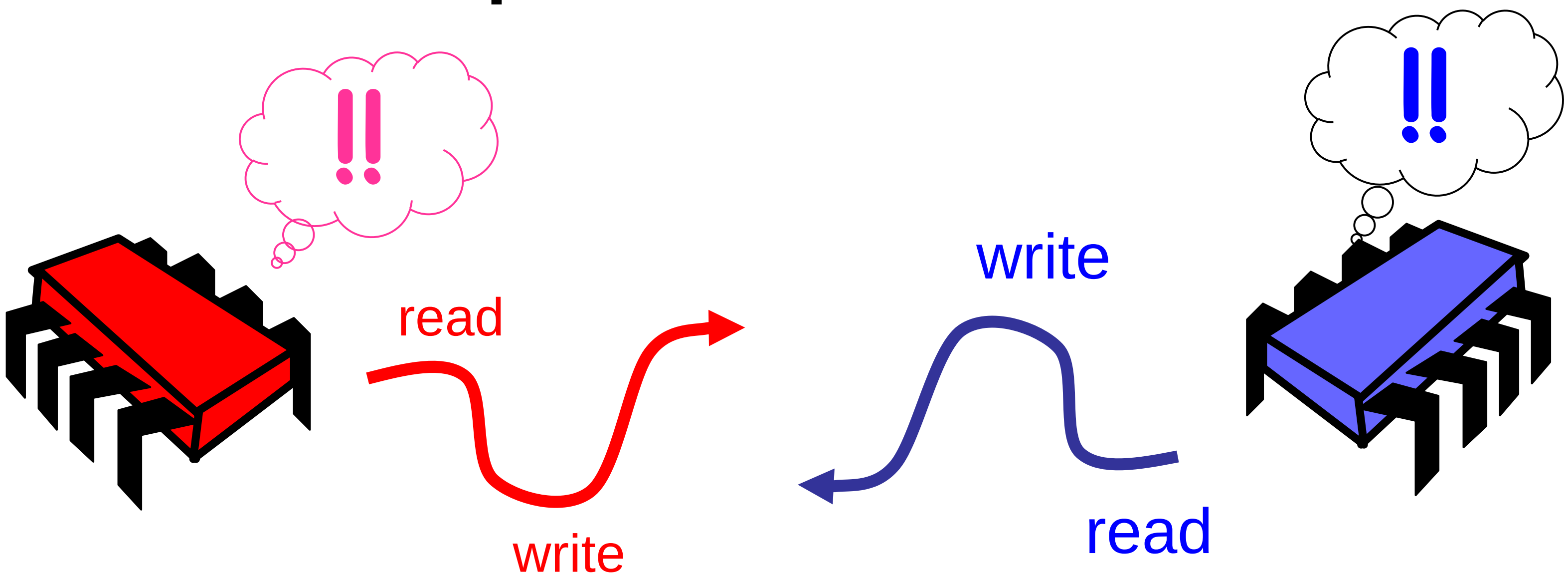
```
temp  = value;
value = temp + 1;
return temp;
```

Race condition

thread 1	thread 2	value
`temp = value`		0
	`temp = value`	0
	`temp = temp + 1`	0
	`Value = temp`	1
`temp = temp + 1`		1
`value = temp`		**1 (not 2!)**

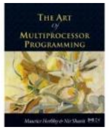
- Result depends on the timing execution of the code
- Can get different result every time!

Is this problem inherent?



What we have here is a **critical section**

If we could only “glue” reads and writes together…i.e. make them atomic…



Hardware Solution

- Read-Modify-Write instructions
- Atomic operations at **CPU level**!
  - Supported in instruction set
  - Example: intel x86 supports *atom_inc*, *atom_dec*, *atom_add*, *atom_sub*
- Built-in functions for atomic memory access
- `type __sync_fetch_and_add (type *ptr, type value, ...)`
- ***`Code example 1`***

Linux Synchronization

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

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 */
unlock
```

Mutex Formal Requirements

- Mutually exclusive
  - Only **one** thread can hold it `lock()`ed – can't have two in same critical section
- Deadlock-free
  - **Some** thread eventually enters critical section
- Starvation-free
  - **Each** thread eventually enters 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

- Creation:

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

- Destruction:

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

Mutex API - Example

```
pthread_mutex_t lock;

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

Locking

Lock:

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

- Acquire lock, block until acquired

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

- Acquire lock once, fail if already locked

Unlock:

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

- Release locked mutex

- ***Code example 2***

Under the Hood

- Known algorithms: **Peterson's**, **bakery**
  - Use a lot of memory (***O(t)*** for ***t*** threads)

- Another suggestion:
  - 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 "locked", no other task can enter critical section

Under the Hood

```
void init() {
    flag = 0;
}
void lock() {
    // busy-wait, loop until value is 0 → unlocked!
    while (flag == 1);
    flag = 1;
}
void unlock() {
    flag = 0;
}
```

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
}
```

Test-and-Set 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;
}
```

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…)

- Without events:
    - Possible deadlock?

Condition Variables (1)

- Allow thread to sleep-wait() on event

```
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)

```c
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);
}
```