Concurrency

An OS is concurrent software, it might be doing many things at once.

On a multi-core system events happen simultaneously

But even with a single CPU we have concurrency

e.g., Processes overlap their executions with each other and with I/O

A process can be interrupted between any two instructions

What’s the problem?

`totreads++` takes three instructions:

1. Read from memory into register
2. Increment the register value
3. Write from register back to memory

…and an interrupt can happen between any two instructions!

One Possible Scenario

Process 1	**Process 2**
call read()	
…	
Read val from memory	
<interrupt>	
.	call read()
.	…
.	read val from memory
increment val	increment val
write val to memory	write val to memory
…	<interrupt>

This is a bug!

It’s called a “data race”, one type of “race condition”

Concurrent, unprotected read/write of shared memory

Much more prevalent with multi-core CPUs

An OS is inherently concurrent

It can happen within user processes too

It makes the code “indeterminate” while we expect this code to be “deterministic”

Here’s a possible (untested) solution

```
int totreads = 0;   // global
int sys_read(void) {
  ...
  suspend_all_interrupts();  // pseudocode
  totreads++;
  allow_interrupts();  // pseudocode
  ...
}
```

Now our three instructions can execute unmolested!

Atomicity, Critical Sections

Disabling interrupts makes our three little instructions “atomic”

Atomic – “as a unit”, “all or none”

Now we can build “critical sections”, sections of code in which shared data structures may be updated and read without data races

Critical Sections, Mutual Exclusion

One way to support Critical Sections is with Mutual Exclusion

Mutual Exclusion (mutex): guarantee that if one schedulable entity (one process, one thread) is executing within a critical section, then all others will be prevented from doing so.

Disabling interrupts is one way to do it!

Issues w Disabling Interrupts

1. doesn’t work on multi-core
2. Some interrupts can’t be masked
3. Must be done in privileged mode
4. It is a blunt instrument
5. Poor performance for user-level processes

Still, it’s a common tactic within OSs. `grep pushcli *.c` to see some xv6 examples.

Review

Indeterminate: a program consisting of one or more race conditions; the output of the program varies run to run. Non-deterministic. Usually (but not always) bad.

Mutual Exclusion: a guarantee that only a single entity can enter a critical section, thus avoiding race conditions.

Review

Atomic: as a unit. All or none. If a system can make a critical section atomic then it can achieve mutual exclusion.

Masking/Disabling Interrupts: one technique used by computer systems to achieve atomicity and provide mutual exclusion for critical sections, thereby avoiding race conditions and ensuring deterministic execution.

Threads

- What is a thread?
- How is it different from a process?
- Posix Threads - pthreads
  - Creating, running, joining and destroying
  - Locks
  - Condition Variables (next week)
- Lab

Review: Process

Is an instance of a program

Is a Virtualization of a CPU

Has an Address Space

Has a set of open file descriptors

Has a CPU state (e.g., registers)

Has scheduling state (running, ready, …)

Has lots of other state

Is scheduled by OS

Is separated/protected from other processes

Threads

Along the way, we discovered that concurrency was useful **within** a process

- Signals – software interrupts
- GUIs
- RDBMSs

So we invented threads

- and there were many varieties!

Threads

Is an execution path within a program

Shares an Address Space

Shares a set of open file descriptors

Has a CPU state (e.g., registers)

Has scheduling state (running, ready, …)

Has very little other state

Is scheduled by OS (usually)

Is not protected from threads

User-level vs Kernel-level threads

Historically, many “threads packages” were developed in user-space. (a.k.a., “LWP”)

Today, usually the kernels schedule them.

Usually, not always.

New Concept: “schedulable entity” which means “process or thread”

Threads and Processes

Processes can contain threads

All threads within a process share the process's address space, file descriptors, resources

Threads have their own stack, registers, scheduling state

TCB – thread control block

POSIX Threads (pthreads)

A standard threads interface

Can be implemented various ways.

Linux: NPTL (native posix thread lib) came from RedHat (2003)

simple pthreads example

```
#include <pthread.h>
…
Main(int argc, char **argv) {
  …
  pthread_t t1;
     = pthread_create(@t1,NULL,(void) *func(), void *arg);
  …
  pthread_join(thread1, NULL);
  …
}
```

other basic lifecycle operations

```
void pthread_exit(void *status);
pthread_t pthrad_self(void);
pthread_attr_*  // for manipulating thread attributes
int pthread_detach(pthread_t thread);
pthread_cleanup* // various ways to handle thread cleanup
```

What about critical sections?

Mutex Locks!

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int totReads;

int sys_read(void) {
  pthread_mutex_lock( &mutex1 );
  totReads++;    // critical section
  pthrad_mutex_unlock( &mutex1 );
}
```

pthread_mutex

How is that implemented?

look inside the implementation to find the use of the Intel **xchg** instruction.

xchg: swap the contents of a memory location with a register value

Not expressable in C so you use assembly

pthread_mutex (pseudo code)

```
int lockval = 0;   // this is global, 0 == “unlocked”

lock() {
  register int regval = 1;
  while (xchg (lockval, regval));  // spin
}

unlock() {
   lockval = 0;
}
```

other mutex operations

```
pthread_mutex_destroy()
pthread_mutex_trylock()    // avoid spinning
pthread_mutex_timedlock()  // time out
```