Pthreads

- Each OS has its own thread package with different Application Programming Interfaces ⇒ poor portability.
- Pthreads
  - A POSIX standard API for thread management and synchronization.
  - API specifies behavior of the thread library, not the implementation.
  - Commonly supported in UNIX operating systems.

Thread Creation

```
int pthread_create
  (pthread_t *new_id,
  const pthread_attr_t *attr,
  void *(*func) (void *),
  void *arg)
```

- new_id: thread's unique identifier
- attr: ignore for now
- func: function to be run in parallel
- arg: arguments for function func

Example of Thread Creation

```
void *func(void *arg) {
  int      *I=arg;
  …..
}

void main()
{
  int X;   pthread_t    id;
  ….
  pthread_create(&id, NULL, func, &X);
  …
}
```


Pthread Termination

```
void pthread_exit(void *status)
```

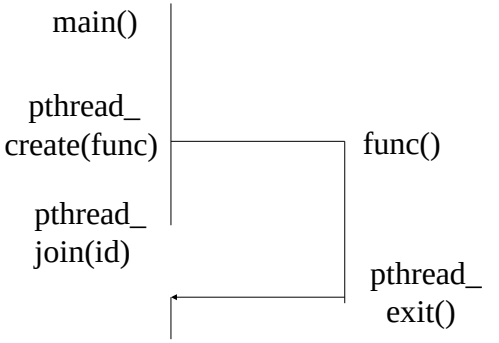
- Terminates the currently running thread.
- Is implicit when the function called in pthread_create returns.

Thread Joining

```
int pthread_join(
pthread_t new_id,
void **status)
```

- Waits for the thread with identifier new_id to terminate, either by returning or by calling pthread_exit().
- Status receives the return value or the value given as argument to pthread_exit().

Example of Thread Creation



Contention Scope

- Process contention scope – thread library schedules user threads onto light-weight processes (kernel-level thread)
  - Use priority as defined by user – no preemption of threads with same priority
- System contention scope – compete with all tasks and schedule kernel thread on a physical CPU
- pthreads: PTHREAD_SCOPE_PROCESS, PTHREAD_SCOPE_SYSTEM
  - pthread_attr_setscope
  - pthread_attr_getscope

Pthread Attributes

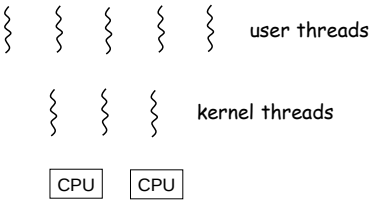
- Pthread_attr_init(pthread_attr_t *attr), destroy – initializes attr to default value
  - Scope – pthread_attr_setscope (&attr, SCOPE)
  - Stack size – pthread_attr_getstacksize, pthread_attr_setstacksize
  - Priority
  - Joinable or detached

User/Kernel Threads

- User threads
  - Thread data structure is in user-mode memory
  - scheduling/switching done at user mode
- Kernel threads
  - Thread data structure is in kernel memory
  - scheduling/switching done by the OS kernel
- Benefits of user threads
  - lightweight – less context switching overhead
  - flexibility – allow application-controlled scheduling
- Problems of user threads
  - can't use more than one processor
  - oblivious to kernel events, e.g., all threads in a process are made to wait when only one of them does I/O

Mixed User/Kernel Threads

- M user threads run on N kernel threads (M≥N)
  - N=1: pure user threads
  - M=N: pure kernel threads
  - M>N>1: mixed model



Solaris/Linux Threads

- Solaris
  - supports mixed model
- Linux
  - No standard user threads on Linux
  - Processes are treated similarly with threads (both called tasks)
  - Processes are tasks with exclusive address space
  - Tasks can also share the address space, open files, …

Issues with the Threading Model

- Thread-local storage – what about globals?
- Stack management
- Interaction with fork and exec system calls
  - Two versions of fork?
- Signal handling – which thread should the signal be delivered to?
  - Synchronous
  - All
  - Assigned thread
  - Unix: could assign a specific thread to handle signals
  - Windows: asynchronous procedure calls, which are thread-specific

SYNCHRONIZATION IN THE LINUX KERNEL

Examples of OS Kernel Synchronization

- Two processes making system calls to read/write on the same file, leading to possible race condition on the file system data structures in OS
- Interrupt handlers put I/O data into a buffer queue that might be retrieved by application-initiated I/O system calls

OS Kernel Structure for Synchronization

- OS is divided into two parts:
  - upper part (serving application requests): system call, exception
  - lower part (serving hardware device requests): interrupt handling
- Upper part runs in process/thread context
  - resource accounting to corresponding process/thread
  - running on a kernel stack usually associated with the corresponding process/thread control block
- Lower part runs in a separate interrupt context
  - resource accounting to who?
  - running in a separate (often dedicated) kernel interrupt stack
- Blocking behaviors:
  - Upper part may block (yield CPU), interleave with others
  - Lower part does not block, must run atomically (one by one) – interrupt handlers typically run with other interrupts disabled
- Preemption/priority:
  - A lower part interrupt handler may preempt an upper part system call processing, but not vice versa

OS Kernel Synchronization

- Available mechanisms:
  - disabling interrupts
  - spin_lock (busy waiting lock)
  - blocking synchronization (mutex lock, semaphore, …)
- Synchronization between upper part kernel "threads"
  - typically blocking synchronization
  - Spin lock if critical section short (only useful on multiprocessor)
- Synchronization between an upper part kernel "thread" and a lower part interrupt handler:
  - if blocking synchronization: block only at upper part, never lower part (possible in semaphore)
  - Spin lock may be used (only useful on multiprocessor)
  - the upper part should disable interrupt before entering critical section

A Little More on OS Kernel Structure

- Lower part interrupt handlers do not block
  - interrupt handlers typically run with other interrupts disabled
- This can be a problem when interrupt handlers do more and more work
- In modern OSes, interrupt handlers typically defer some work to later (interruptible contexts)
  - soft irqs in Linux

Preemptible Kernel

- Preemptible kernel
  - One in which a process switch may occur at any point when a process is executing in kernel mode
  - Requires the re-entrant property
    - Several processes may be executing in kernel mode at the same time
    - Use either re-entrant functions (ones that don't modify glabal variables) or thread-safe functions

Synchronization in Linux

Technique	Description	Scope
Per-CPU variables	Duplicate a data structure among the CPUs	All CPUs
Atomic operation	Atomic read-modify-write instruction to a counter	All CPUs
Memory barrier	Avoid instruction reordering	Local / All CPUs
Spin lock	Lock with busy wait	All CPUs
Semaphore	Lock with blocking wait (sleep)	All CPUs
Seqlocks	Lock based on an access counter	All CPUs
Local interrupt disabling	Forbid interrupt handling on a single CPU	Local CPU
Local softirq disabling	Forbid deferrable function handling on a single CPU	Local CPU
Read-copy-update (RCU)	Lock-free access to shared data structures through pointers	All CPUs

Per-CPU Variables

- An array of data structures, one element per CPU
- Ensure that element falls on a unique cache line
- Caveats?
  - Still require disabling preemption on a single CPU
  - Prone to race conditions because of the above

Atomic Operations

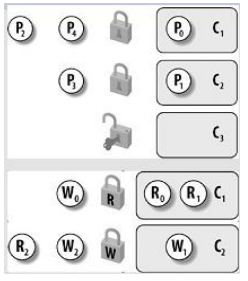
- Read-modify-write
  - Use specific atomic instructions or lock prefix on x86
- Ensures that operations are not interleaved with those by other threads
  - Locally ensures that process will not get context switched between read and write in read-modify-write (single opcode)

Memory Barriers

- Prevent compiler reordering (e.g., reordering for optimized register use)
  - Optimization barrier
- Prevent hardware reordering
  - Memory barrier
  - Instructions that operate on I/O, or are "locked" (on x86 machines)
  - Writes to control registers
  - … a few others

Spin Locks (+ R-W spinlocks)

- Ensure that code executing spin lock is non-blocking
- spinlock_t (uses xchgb on x86)
- R-W spin locks
  - Uses an atomic decrement and subtract with multiple values for the lock



Sequence locks

- Higher priority to writers
- Seqlock_t consists of two fields – spinlock and an integer sequence
- Reads – read sequence before and after

```
do {
    seq = read_seqbegin(&seqlock);
    … critical section …
} while (read_seqretry(&seqlock, seq);
```

- Writes – acquire spinlock, increase sequence by 1 on entry; increase sequence by 1 and release spinlock on exit
  - write_seqlock()
  - write_sequnlock()

Read-Copy Update

- Non-blocking form of synchronization
- Reader reads in place
  - Data structure must be dynamically allocated and referenced using pointers
  - Disable preemption
- Copy data structure to write; switch pointers (requires memory barriers to ensure that data structure modification precedes pointer modification)
- Old copy can be freed only after all potential readers have unlocked – special function to free old copy

Semaphores

- struct semaphore
  - atomic_t count
  - wait (wait queue list address)
  - Sleepers (count to indicate processes are sleeping)
- To be used only by functions allowed to block
- Read/write semaphores
- Mutexes (binary semaphores)

Synchronization in Linux

Goal: Maximize concurrency

- Per-CPU variables to avoid synchronization
- Atomic variables (non-blocking)
- Read-copy-update (non-blocking)
- Sequence locks (writer-prioritized reader/writer locks)
- Spin-locks – basic, r/w (blocking)
- Semaphores (sleeping)
- Local interrupt disabling

Disclaimer

- Parts of the lecture slides were derived from those by Kai Shen, Willy Zwaenepoel, Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Andrew S. Tanenbaum, and Gary Nutt. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).