4/12/24 Locks & Monitors

→ Types of Locks

→ spinlock: spins on the CPU while the lock is busy

↳ wastes the CPU

↳ why use this at all on a single core system?

→ sometimes you can't block! (interrupt handler context)

→ still good for short critical section

→ locks used by also disables interrupts

→ sleeplock: blocks/sleeps while the lock is busy

↳ context switch overhead

(scheduling, switching VAS)

→ Lock Granularity

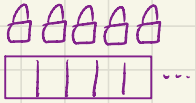
→ how much shared data should a lock protect?

→ a single lock for an entire array (coarse grained)

→ does provide safe access, simple, easy to perform multi-entries ops.

→ no concurrent access to the array

→ one lock per array entry (fine grained)



→ allows for concurrent independent ops on each entry

→ higher locking overheads, easier to get deadlocks

⇒ can go finer (protect part of a struct)

```
struct PCB {
    pid
    :
    fd table
    scheduling state
}
```

only accessed by the process

↓ accessed by processes (exit, wait) & the scheduler

disable interrupts
→ avoids preemption
→ provides mutual exclusion on a single core
→ privileged instr. not general

xk inode case study

```
struct {
  struct spinlock lock;
  struct inode inode[NINODE];
  struct inode inodefile;
} icache;
```

↑ acquires global spinlock

```
struct inode *idup(struct inode *ip) {
  acquire(&icache.lock);
  ip->ref++;
  release(&icache.lock);
  return ip;
}
```

Spinlock protects ref field of every single inode struct.

```
// in-memory copy of an inode
struct inode {
  uint dev;  // Device number
  uint inum; // Inode number
  int ref;   // Reference count
  int valid; // Flag for if node is valid
  struct sleeplock lock;

  // copy of disk inode (see fs.h for details)
  short type;
  short devid;
  uint size;
  struct extent data;
};
```

acquires per inode sleeplock

```
int concurrent_readi(struct inode *ip, char *dst, uint
  int retval;

  locki(ip);
  retval = readi(ip, dst, off, n);
  unlocki(ip);
```

Monitors

→ design pattern & synchronization construct that coordinate threads based on events

→ a monitor = a lock + resource state(s) + condition variables

a lock: doesn't matter what type of lock, protects accesses to conditions & condvars

resource state(s): track states used for determining conditions; e.g. child→state == ZOMBIE

condition variables: manages waiters of a condition

→ **wait** [xk: sleep]

put the calling thread to waiter list
blocks the thread & releases the lock atomically,
reacquires the lock & then returns upon unblock

→ **Signal**

wake up a waiter (Blocking → ready),
remove from the waiter list

→ **Broadcast** [xlc: Wakeup]

wake up all waiters, used when the condition may enable multiple waiters (e.g. N threads need to wake up at time x)

Basic Pattern

function can be accessed by many threads
access to condition & condvars must be protected w/ lock!

```
consume() {
    lock.acquire();
    while (!condition) {
        cv_wait(lock);
    }
    // consume condition
    condition = False;
    lock.release();
}
```

cv_wait() { atomically releases the lock & blocks, <other threads cause changes to shared states> reacquires the monitor lock before returning }

☆ if block & lock release is not atomic, we will have
lock_release();
<other threads may run & change states>
cv_wait();
lock_acquire();
which suffers from time of check to time of use: wait is done bc cond is false, but that might no longer be true!

```
produce() {
    lock.acquire();
    // generate condition
    condition = True;
    cv_signal();
    lock.release();
}
```

updates waiter's scheduling state to ready

☆ Spurious Wakeups: when a thread wakes up, the condition might not be true!

MESA Monitor: has this semantic, a new thread may acquire the lock before the woken up waiter

Hoare monitor: the woken up waiter is guaranteed the lock next (after signaling thread releases)