Asynchronous Completion

- 7C Asynchronous Completion
- 7F Asynchronous Completion Operations
- 7G Implementing Asynchronous Completion

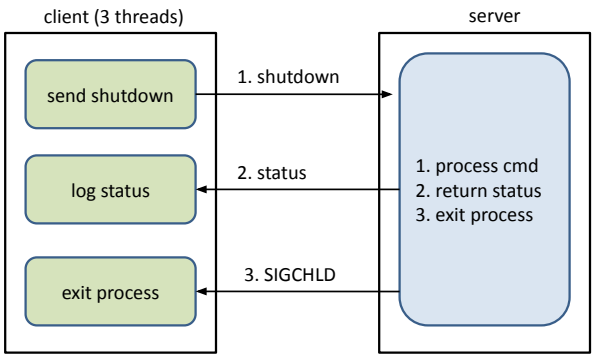
Problem 2: asynchronous completion

- most procedure calls are synchronous
  - we call them, they do their job, they return
  - when the call returns, the result is ready
- some operations are independent of our code
  - release of a lock, held by another process
  - completion of a file I/O operation
  - response to a network request
  - passage of some period of time
- we call such completions a-syn-chronous

Why We Wait

- We await completion of non-trivial operations
  - data to be read from disk
  - a child process to be created
- We wait for important events
  - a response/notification from another process
  - an out-of-band error that must be handled
- We wait to ensure correct ordering
  - B cannot be performed until A has completed
  - if A precedes B, B must see the results of A

Correct Ordering



"Surely the final status message will be received and processed before the SIGCHLD causes the client to shut down! "

Approaches to Waiting

- spinning … "busy waiting"
  - works well if event is independent and prompt
  - wasted CPU, memory, bus bandwidth
  - may actually delay the desired event
- yield and spin … "are we there yet?"
  - allows other processes access to CPU
  - wasted process dispatches
  - works very poorly for multiple waiters
- either may still require mutual exclusion

Spinning Sometimes Makes Sense

1. awaited operation proceeds in parallel
   - a hardware device accepts a command
   - another CPU releases a briefly held spin-lock
2. awaited operation guaranteed to be soon
   - spinning is less expensive than sleep/wakeup
3. spinning does not delay awaited operation
   - burning CPU delays running another process
   - burning memory bandwidth slows I/O
4. contention is expected to be rare
   - multiple waiters greatly increase the cost

The Classic "spin-wait"

```
/* set a specified register in the ZZ controller to a specified value   */
zzSetReg( struct zzcontrol *dp, short reg, long value ) {
        while( (dp->zz_status & ZZ_CMD_READY) == 0);
                /* it may take a few ns to process the last set  */
        dp->zz_value = value;
        dp->zz_reg = reg;
        dp->zz_cmd = ZZ_SET_REG;
}

/* program the ZZ for a specified DMA read or write operation              */
zzStartIO( struct zzcontrol *dp, struct ioreq *bp ) {
        zzSetReg(dp, ZZ_R_ADDR, bp->buffer_start);
        zzSetReg(dp, ZZ_R_LEN, bp->buffer_length);
        zzSetReg(dp, ZZ_R_CMD, bp->write ? ZZ_C_WRITE : ZZ_C_READ );
        zzSetReg(dp, ZZ_R_CTRL, ZZ_INTR + ZZ_GO);
}
```


Correct Completion

- Correctness
  - no lost wake-ups
- Progress
  - if event has happened, process should not block
- Fairness
  - no un-bounded waiting times (multiple waiters
- Performance
  - cost of waiting
  - promptness of resuming
  - minimal spurious wake-ups

Spinning and Yielding

- yielding is a good thing
  - avoids burning cycles busy-waiting
  - gives other tasks an opportunity to run
- spinning and yielding is not so good
  - which process runs next is random
  - when yielder next runs is random
- Progress: potentially un-bounded wait times
- Performance: each try is wasted cycles

Sleep/Wakeup Operations

- sleep (e.g. pthread_cond_wait)
  - block caller until condition has been posted
- wakeup (e.g. pthread_cond_signal)
  - post condition and awaken blocked waiter(s)
- potential problems:
  - race conditions between sleep and wakeup
    - wakeup called before (or during) sleep
  - *spurious* wakeups
    - woken up, but event is not (currently) available

Race: wakeup called before sleep

- model #1 (e.g. pthread condition variables)
  - purely a signaling mechanism
  - client responsible for checking condition

```
pthread_mutex_lock(&mutex);
while( !condition )
  pthread_cond_wait(&condvar, &mutex);
pthread_mutex_unlock(&mutex);
```

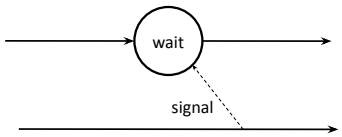
- model #2 (e.g. semaphores)
  - return guarantees condition has been satisfied
  - if condition already satisfied, caller will not block

Spurious Wakeups

- waking up does not mean condition satisfied
  - perhaps multiple processes were woken up
  - perhaps you were woken up for another reason
  - perhaps another process got to resource first
- check/sleep should be done in a loop
  - after each wakeup, check condition again
- spurious wakeups are a minor cost/irritation
- lost wakeups are a serious problem

Condition Variables

- create a synchronization object
  - associate that object with a resource or request
  - requester blocks awaiting event on that object
  - upon completion, the event is "posted"
  - posting event to object unblocks the waiter



pthread_cond_{signal,wait}

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
…
// waiter
pthread_mutex_lock(&lock);
while (ready == 0)
    pthread_cond_wait(&cond, &lock);
…
pthread_mutex_unlock(&lock);
```

```
// signaller
pthread_mutex_lock(&lock);
ready = 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&lock);
```


Evaluating pthread_cond_signal/wait

- Effectiveness/Correctness
  - good (if used properly)
- Progress
  - good (if used properly)
- Fairness
  - who gets resource is random
- Performance
  - good for single consumers
  - potential spurious wakeups w/more consumers

Waiting Lists

- Who wakes up when a CV is signaled
  - pthread_cond_wait … at least one blocked thread
  - pthread_cond_broadcast … all blocked threads
- this may be wasteful
  - if the event can only be consumed once
  - many processes wake and try, most will fail
  - potentially unbounded waiting times
- a waiting queue would solve these problems
  - post wakes up one (first, highest priority) client

Progress vs. Fairness

- consider …
  - P2: lock()
  - P1: lock(), park()
  - P2: unlock(), unpark()
  - P3: lock()
- progress says:
  - it is available, P3 gets it
  - P1 gets spurious wakeup
- fairness says:
  - FIFO, P3 gets in line
  - and a convoy forms

```
void lock(lock_t *m) {
    while(true) {
      while (TestAndSet(&m->guard, 1) == 1);
      if (!m->locked) {
            m->locked = 1;
            m->guard = 0;
            return;
      }
      queue_add(m->q, me);
      m->guard = 0;
      park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1);
    m->locked = 0;
    if (!queue_empty(m->q))
      unpark(queue_remove(m->q);
    m->guard = 0;
}
```


Arpaci: park() and unpark()

You may have missed their definitions:

- park()
  - mark this process no-longer-ready and yield
  - used to sleep, awaiting a later wake-up
  - if process is hyper-awake, do not block it
- unpark(process)
  - clear the not-ready indication for process
  - if process not yet blocked, mark it hyper-awake
  - used to awaken a parked process

Evaluating Sleep w/Waiting Lists

- Effectiveness/Correctness
  - good (if used properly)
- Progress
  - good … if we allow cutting in line
- Fairness
  - good … unless we allow cutting in line
- Performance
  - good (with possible spurious wakeups)

Locking and Waiting Lists

- Spinning for a lock is usually a bad thing
  - locks should probably have waiting lists
- a waiting list is a (shared) data structure
  - implementation will likely have critical sections
  - which may need to be protected by a lock
- This seems to be a circular dependency
  - locks have waiting lists
  - which must be protected by locks
  - what if we must wait for the waiting list lock?

Race Condition within Sleep

```
void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1);
    if (!m->locked) {
      m->locked = 1;
      m->guard = 0;
    } else {
      queue_add(m->q, me);
      m->guard = 0;

      park();
    }
}
```

```
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1);
    if (queue_empty(m->q))
      m->locked = 0;
    else
      unpark(queue_remove(m->q);
    m->guard = 0;
}
```


(sleep/wakeup races)

- possibility of long spins or deadlock
  - interrupt comes in while guard is held
  - ISR tries to wake-up the waiting list
- possibility of missed wakeup
  - wakeup is sent before blockee can sleep
  - blockee sleeps, having missed the wakeup
- solutions (may require OS assistance)
  - disable interrupts in this critical section
  - unpark() makes process hyper-awake

Reading and Assignments

Reading:

- Arpaci C30.2-3 … producer/consumer problems
- Arpaci C31 … deadlock
- Java Synchronization, Java Intrinsic Locks
- Monitors

Projects:

- start looking at project 4B (embedded system sensor I/O)

Supplementary Slides

Synchronization Objects

- combine exclusion and (optional) waiting
- operations implemented safely
  - with atomic instructions
  - with interrupt disables
- exclusion policies (one-only, read-write)
- waiting policies (FCFS, priority, all-at-once)
- additional operations (queue length, revoke)

Spin Waiting For Asynchronous Completions

- Wastes CPU, memory, bus bandwidth
  - Each path through the loop costs instructions
- May actually delay the desired event
  - One of your cores is busy spinning
  - Maybe it could be doing the work required to complete the event instead
  - But it's spinning . . .

Spin-Waits Revisited

- spin-waits await asynchronous completions
  - but they do so by busy-waiting

    while (event_not_ready) ;
- sleep/wake-up is almost always better
  - fewer wasted cycles and faster response
  - these are software completion mechanisms and there are hardware-related situations where they don't work (or don't make sense)
- there are cases where it makes sense to spin
  - very briefly for events originating outside our CPU

Spin-waits: when to use them

- when the event does not come from our CPU
  - so spinning will not delay the completion
- and waiting time guaranteed to be very brief
  - fewer cycles than would be required to go to sleep
- examples:
  - waiting a few μ-seconds for hardware to come ready
    - **IF** it is guaranteed to be come back promptly
  - waiting for another CPU to release a lock
    - **IF** critical section is very short (e.g. 1 digit # of instructions)
    - **IF** interrupts are disabled so preemption is impossible
- almost never appropriate in user-mode code