CS 162: Operating Systems and Systems Programming

Lecture 7: Synchronization Operations

September 19, 2019

Instructor: David Culler

https://cs162.eecs.berkeley.edu

Read: A&D Ch 5

What's a Process?

- The execution instance of a program
- Comprised of
  - One or more threads of execution
    - Processor registers, stack, OS support
  - A virtual address space
    - Page table maps resident pages to memory
  - Set of open file descriptors & buffers
    - User gets handle, OS holds the real thing
  - Whatever else the OS needs to load, run, manage, and terminate it

Grounding Demo

```
(base) CullerMac19:fa19 culler$ ps -al
  UID   PID  PPID        F CPU PRI NI       SZ    RSS WCHAN     S             ADDR TTY           TIME CMD
    0 55813 55812     4106   0  31  0  4336432     12 -        Ss               0 ttys000    0:00.34 login -pf culler
  501 55814 55813     4006   0  31  0  4325240    948 -        S                0 ttys000    0:00.07 -bash
    0 61830 55814     4106   0  31  0  4270584   1120 -        R+               0 ttys000    0:00.00 ps -al
(base) CullerMac19:fa19 culler$ jobs
(base) CullerMac19:fa19 culler$ top
```

```
Processes: 395 total, 2 running, 393 sleeping, 1868 threads
Load Avg: 1.55, 1.32, 1.23  CPU usage: 0.96% user, 1.20% sys, 97.82% idle  SharedLibs: 280M resident, 48M data, 28M linkedit.
MemRegions: 116759 total, 3602M resident, 119M private, 1370M shared. PhysMem: 10G used (2601M wired), 6316M unused.
VM: 1763G vsize, 1372M framework vsize, 28191649(0) swapins, 29789745(0) swapouts. Networks: packets: 24399684/12G in, 10589805/3743M out.
Disks: 5514555/198G read, 6440971/229G written.

PID    COMMAND      %CPU TIME     #TH    #WQ  #PORT MEM    PURG   CMPRS  PGRP  PPID  STATE    BOOSTS          %CPU_ME %CPU_OTHRS UID  FAULTS    COW    MSGSENT
61839  screencaptur 0.1  00:00.19 5      3    192   13M    0B     0B     61839 1     sleeping *0[127+]        0.00000 0.03601    501  8500      352    2717+
61838  screencaptur 1.4  00:00.32 2      1    54    3236K+ 620K   0B     383   383   sleeping *0[1]           0.03601 0.00000    501  15815+    207    3234+
61837  top          3.5  00:00.95 1/1    0    27    6024K  0B     0B     61837 55814 running  *0[1]           0.00000 0.00000    0    8675+     107    435408+
61836  QuickLookSat 0.0  00:00.04 2      1    46    3892K  0B     0B     61836 1     sleeping  0[2]           0.00000 0.00000    501  4470      196    201
61835  quicklookd   0.0  00:00.06 4      1    95    3544K  72K    0B     61835 1     sleeping  0[2]           0.00000 0.00000    501  4562      222    427
61833  mdworker_sha 0.0  00:00.05 3      1    63    3384K  0B     0B     61833 1     sleeping *0[1]           0.00000 0.00000    501  4034      207    662
61828  Google Chrom 0.0  00:00.08 13     1    109   13M    4096B  0B     54320 54320 sleeping *0[4]           0.00000 0.00000    501  9102      1771   401
61821  CoreServices 0.0  00:00.08 3      1    169   4580K  0B     0B     61821 1     sleeping *0[1]           0.00000 0.00000    501  5023      264    1131+
61818  mdworker_sha 0.0  00:00.03 3      1    63    3316K  0B     0B     61818 1     sleeping *0[1]           0.00000 0.00000    501  4108      205    631
61806  Google Chrom 0.0  00:00.86 14     1    150   30M    4096B  0B     54320 54320 sleeping *0[5]           0.00000 0.00000    501  20218     1841   9138
61801  Google Chrom 0.0  00:06.06 14     1    141   37M    4096B  0B     54320 54320 sleeping *0[6]           0.00000 0.00000    501  40084     1863   36873+
61769  mdworker_sha 0.0  00:00.12 3      1    59    4164K  0B     0B     61769 1     sleeping *0[1]           0.00000 0.00000    501  7899      221    1303
61768  mdworker_sha 0.0  00:00.13 3      1    59    4292K  0B     0B     61768 1     sleeping *0[1]           0.00000 0.00000    501  7944      221    1371
61767  mdworker_sha 0.0  00:00.13 3      1    59    4204K  0B     0B     61767 1     sleeping *0[1]           0.00000 0.00000    501  7943      221    1383
61766  mdworker_sha 0.0  00:00.12 3      1    59    4092K  0B     0B     61766 1     sleeping *0[1]           0.00000 0.00000    501  7891      221    1315
61762  mdworker_sha 0.0  00:00.05 4      1    52    4452K  0B     0B     61762 1     sleeping *0[1]           0.00000 0.00000    89   6455      189    613
61726  Google Chrom 0.0  00:00.24 15     2    146   16M    4096B  0B     54320 54320 sleeping *0[6]           0.00000 0.00000    501  12886     1827   3841
61623  netbiosd     0.0  00:01.07 7      7    29    2912K  0B     0B     61623 1     sleeping *0[1]           0.00000 0.00000    222  3904      158    159
61621  eapolclient  0.0  00:00.18 3      1    52    3108K  0B     0B     61621 56    sleeping *0[1]           0.00000 0.00000    501  3902      352    445
61571  mdworker_sha 0.0  00:00.72 3      1    63    14M    0B     0B     61571 1     sleeping *0[1]           0.00000 0.00000    501  7201      209    938
61569  BoxEditFinde 0.0  00:00.40 3      1    159   5208K  0B     0B     61569 1     sleeping *0[27]          0.00000 0.00000    501  5410      303    4025+
61565  com.apple.ap 0.0  00:02.83 3      1    349   37M    220K   0B     61565 1     sleeping  0[144]         0.00000 0.00000    501  33441     406    21061
61478  sandboxd     0.0  00:04.03 4      3    66    5776K  0B     0B     61478 1     sleeping *0[1]           0.00000 0.00000    0    241756    169    311257
61423  com.microsof 0.0  00:00.44 2      1    46    3640K  0B     2588K  61423 1     sleeping *0[1]           0.00000 0.00000    0    6620      283    1303
56470  Microsoft Up 0.0  00:03.51 6      2    200-  12M-   128K   4504K  56470 1     sleeping *0[120]         0.00000 0.00000    501  24520     392    9437+
56468  Microsoft Po 0.0  24:10.79 33     12   2078  644M   189M   107M   56468 1     sleeping  0[2207]        0.00000 0.00000    501  18023272  677881 7539121
56389  usbmuxd      0.0  00:00.06 3      1    38    1404K  0B     712K   56389 1     sleeping *0[1]           0.00000 0.00000    213  2287      146    430
56387  zoom.us      0.0  09:26.93 16     1    843   149M   3260K  115M   56387 1     sleeping  0[345]         0.00000 0.00000    501  14397377  1225   2446416
56332  mdworker_sha 0.0  00:00.16 3      1    58    4272K  0B     2816K  56332 1     sleeping *0[1]           0.00000 0.00000    501  6110      188    1425
56331  LookupViewSe 0.0  00:00.73 3      1    173   10M    0B     6152K  56331 1     sleeping  0[246]         0.00000 0.00000    501  16374     365    6213
55956  Google Chrom 0.0  00:44.99 18     2    223   116M   0B     17M    54320 54320 sleeping *0[5]           0.00000 0.00000    501  229433    7289   201102+
55814  bash         0.0  00:00.07 1      0    21    1020K  0B     452K   55814 55813 sleeping *0[1]           0.00000 0.00000    501  3189      1525   137
55813  login        0.0  00:00.34 2      1    31    1252K  0B     1236K  55813 55812 sleeping *0[9]           0.00000 0.00000    0    1810      196    121
55812  Terminal     1.1  00:19.52 8      3    366   53M-   8692K  17M    55812 1     sleeping *0[544]         0.00100 0.00000    501  399265    451    95590+
55605  PerfPowerSer 0.0  00:11.35 2      1    121   9728K  256K   2424K  55605 1     sleeping  0[165]         0.00000 0.00000    0    22444     207    55139
55597  BoxEditFinde 0.0  00:01.75 3      1    159   6128K  0B     2020K  55597 1     sleeping *0[132]         0.00000 0.00000    501  9193      304    16116+
55576  SimulatorTra 0.0  00:01.32 3      1    137   5672K  0B     2132K  55576 1     sleeping *0[112]         0.00000 0.00000    501  7642      255    14687+
55575  com.apple.Co 0.0  00:01.60 3      1    142   7016K  0B     2960K  55575 1     sleeping *0[115]         0.00000 0.00000    501  8765      289    14991+
55520  Google Chrom 0.0  00:01.36 6      1    72    46M    0B     9984K  54320 54320 sleeping *0[4]           0.00000 0.00000    501  31158     1782   6806
55519  Google Chrom 0.0  00:18.44 13     1    151   91M    0B     23M    54320 54320 sleeping *0[7]           0.00000 0.00000    501  150819    1996   107587
```

Recall: Scheduling

- **First-Come First-Served**: Simple, vulnerable to convoy effect
- **Round-Robin**: Fixed CPU time quantum, cycle between ready threads
- **Priority:** Respect differences in importance
- **Shortest Job/Remaining Time First:** Optimal for average response time, but unrealistic
- **Multi-Level Feedback Queue:** Use past behavior to approximate SRTF and mitigate overhead

Impacts of Scheduling on …

- Lot's of attention to algorithmic complexity of operations on the scheduling data structure
  - These queues don't get that long. Otw, buy more hardware
- Interactions of scheduling with memory hierarchy
  - Locality is fundamentally at odds with fairness
  - "Cache / VM / File buffer *affinity*"
- Interactions of scheduling with multiple processors
  - Processor / Core affinity is really about caches
- Memory performance (locality) is critical

System Design …

- Sophisticated policies (often with deep theoretical basis) boil down into simple manipulation of data structures.
- And understanding multi-dimensional interactions

- We'll return to advanced scheduling (with randomness) later in the term

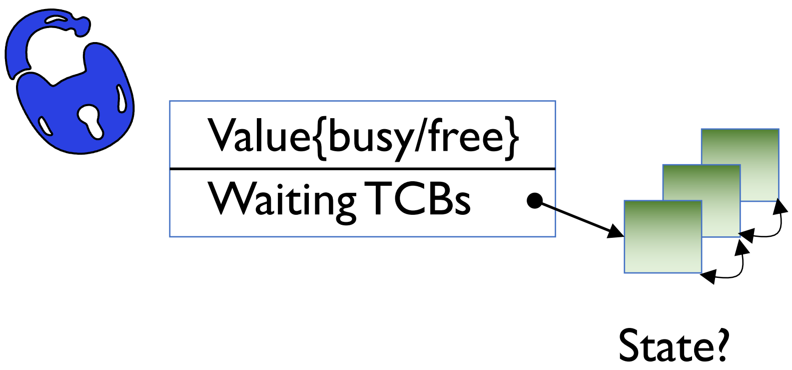
Going back – to a subtle connection between scheduling and synchronization

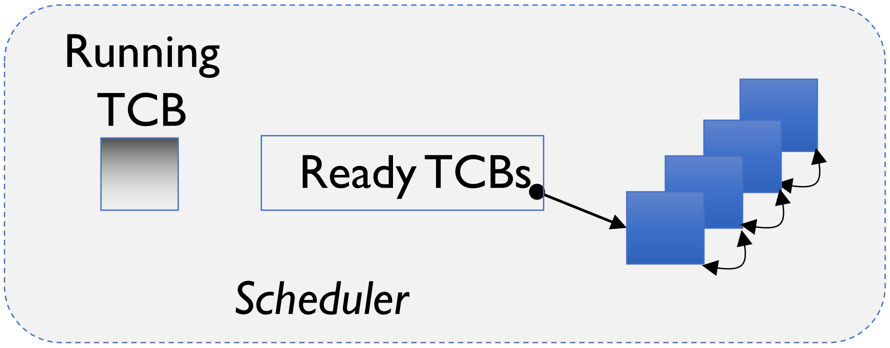
- Why do we need to disable interrupts at all?
  - Avoid interruption between checking and setting lock value
  - Otherwise two threads could think that they both have lock

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

Critical Section

Recall: Basic Lock Implementation






```
Acquire(*lock) {
   disable interrupts;
   if (lock->value == BUSY) {
      put thread on lock's wait_Q
      "i.e, Go to sleep"
       allow a ready thread to run
   } else {
      lock->value = BUSY;
   }
   enable interrupts;
}
```

```
Release(*lock) {
   disable interrupts;
   if (any TCB on lock wait_Q) {
      "i.e., lock busy";
      take thread off wait queue
      Place on ready queue;
   } else {
      lock->value = FREE;
   }
   enable interrupts;
}
```

Reenabling Interrupts When Waiting

enable interrupts

enable interrupts

```
Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    run_new_thread()
} else {
    value = BUSY;
  }
  enable interrupts;
}
```

- Before on the queue?
  - Release might not wake up this thread!
- After putting the thread on the queue?
  - Gets woken up, but immediately switches away

Reenabling Interrupts When Waiting

```
Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    run_new_thread()
  } else {
    value = BUSY;
  }
  enable interrupts;
}
```

enable interrupts →

- Best solution: after the current thread suspends
- How?
  - `run_new_thread()` should do it!
  - Part of returning from `switch()`

How to Re-enable After Sleep()?

- In scheduler, since interrupts are disabled when you call sleep:
  - Responsibility of the next thread to re-enable ints
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts

Thread A		Thread B
. . disable ints sleep		
	context switch →	sleep return enable ints . . . disable int sleep
sleep return enable ints . .	← context switch	

User-level threads?

- Can multiple threads be implemented entirely at user level?
- Most other aspects of system virtualize.

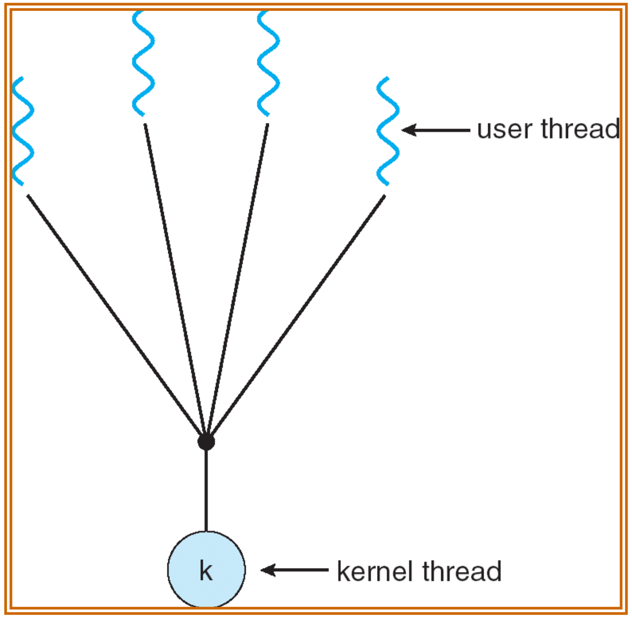
Kernel-Supported Threads

- Threads run and block (e.g., on I/O) independently
- One process may have multiple threads waiting on different things
- Two mode switches for every context switch (expensive)
- Create threads with syscalls

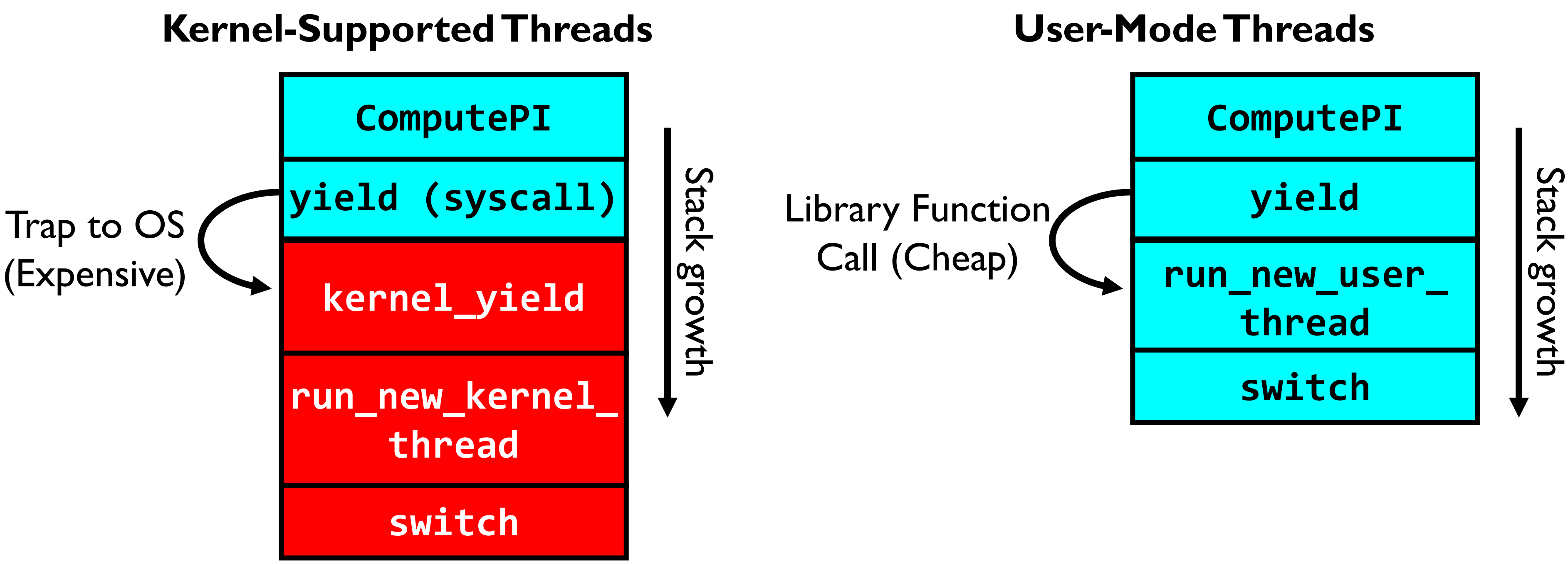
- Alternative: multiplex several streams of execution (at user level) on top of a single OS thread
  - E.g., Java, Go, … (and many many user-level threads libraries before it)

User-Mode Threads

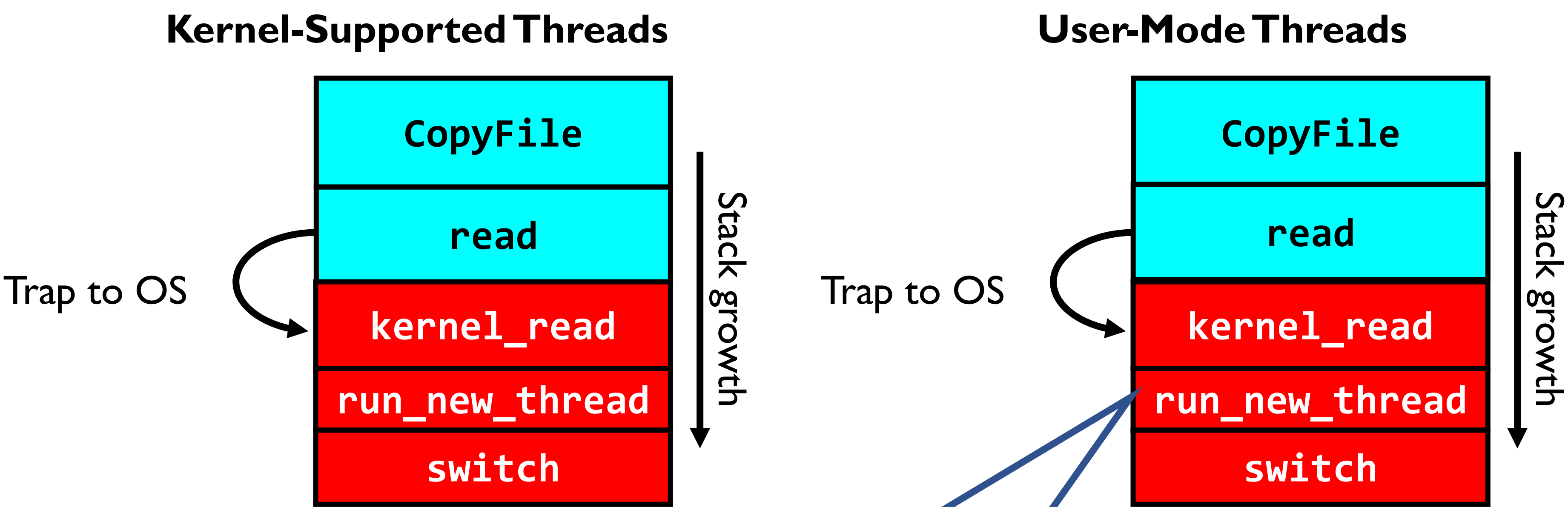
- User program contains its own scheduler
- Several user threads per kernel thd.
- User threads may be scheduled non-preemptively
  - Only switch on `yield`
- Context switches cheaper
  - Copy registers and jump (`switch` in userspace)



Thread Yield



Thread I/O



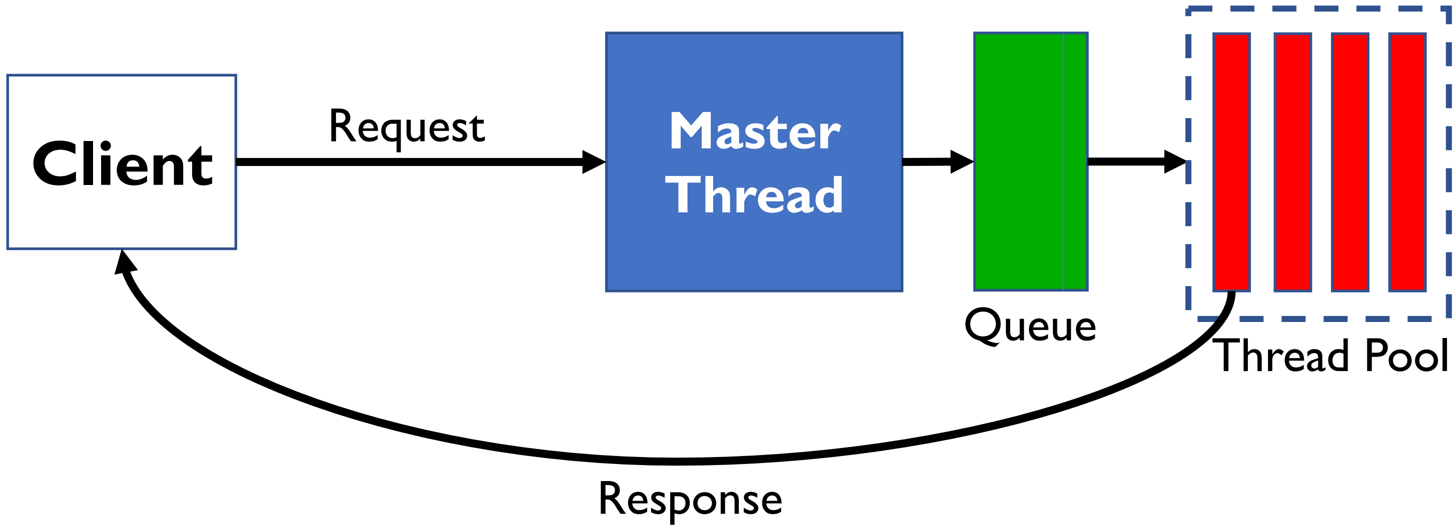
- Selects a new *kernel thread* to run
- Bypassing user-level scheduler

User-Mode Threads: Problems

- One user-level thread blocks on I/O: they all do
  - Kernel cannot adjust scheduling among threads it doesn't know about
- Multiple Cores?
- Can't completely avoid blocking (syscalls, page fault)
- One Solution: *Scheduler Activations*
  - Have kernel inform user-level scheduler when a thread blocks
- Evolving the contract between OS and application.

Recall: Multithreaded Server

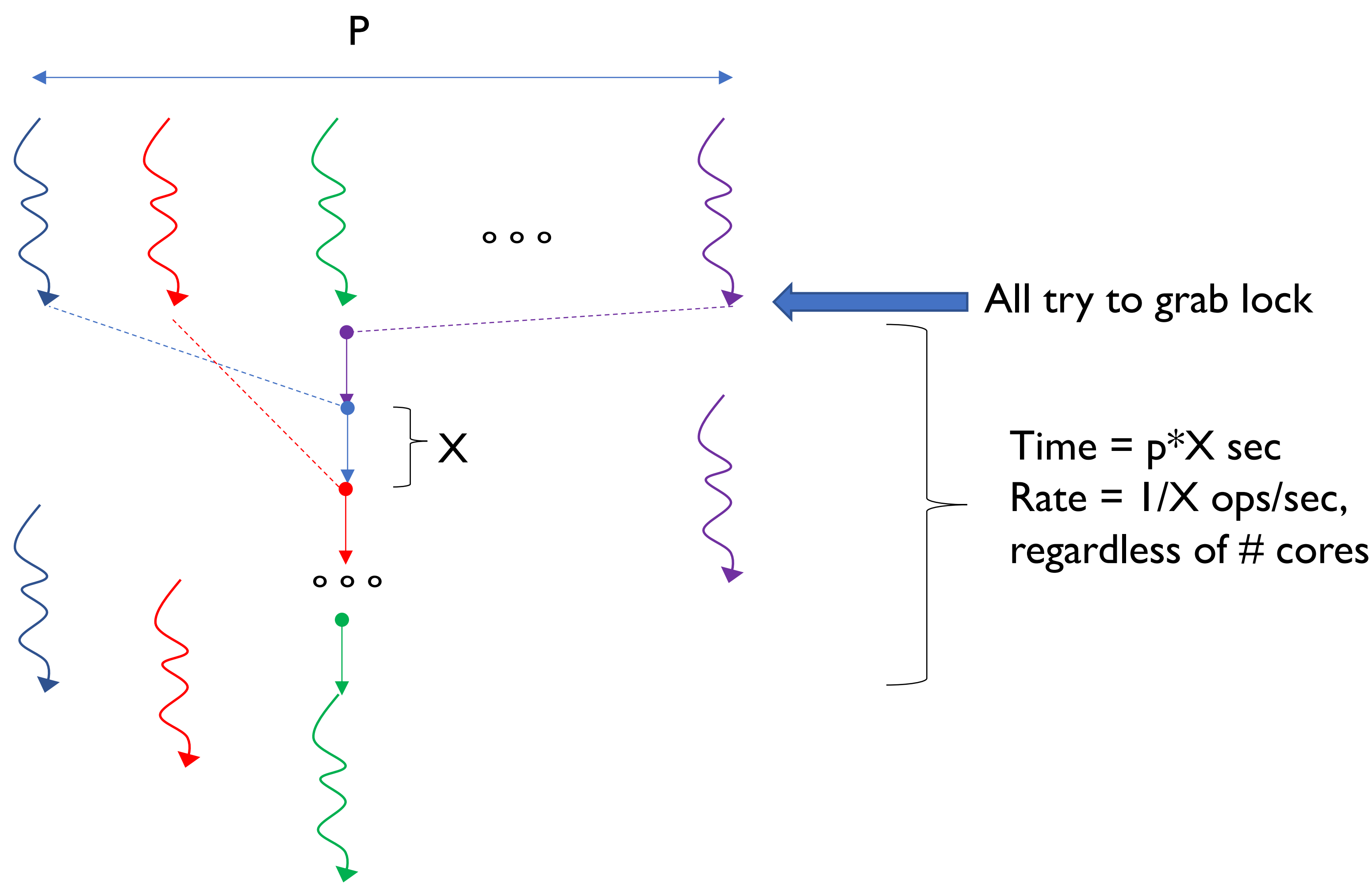
- **Bounded** pool of worker threads
  - Allocated in **advance:** no thread creation overhead
  - **Queue** of pending requests



Simple Performance Model

- Given that the overhead of a critical section is X
  - User->Kernel Context Switch
  - Acquire Lock
  - Kernel->User Context Switch
  - <perform exclusive work>
  - User->Kernel Context Switch
  - Release Lock
  - Kernel->User Context Switch
- Even if everything else is infinitely fast, with any number of threads and cores
- What is the maximum rate of operations that involve this overhead?

Highly Contended Case – in a picture



Back to system performance

More Practical Motivation

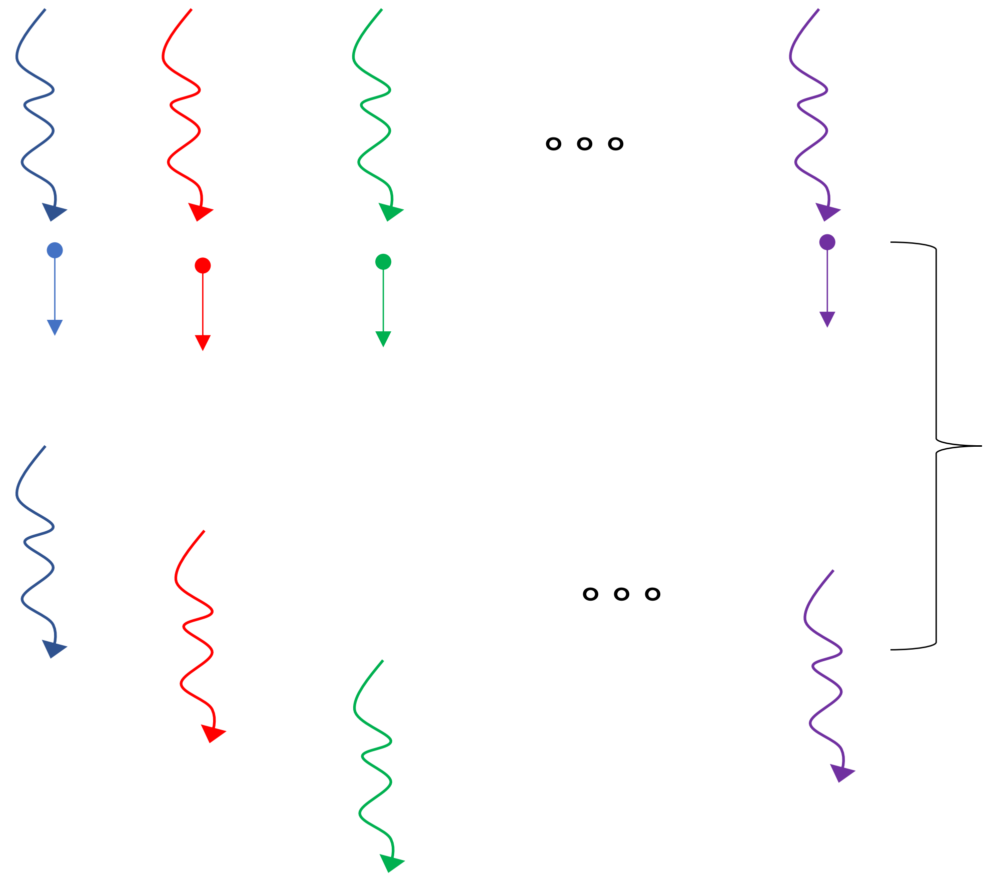
Back to Jeff Dean's "Numbers everyone should know"

Handle I/O in separate thread, avoid blocking other progress

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

- X = 1ms => 1,000 ops/sec

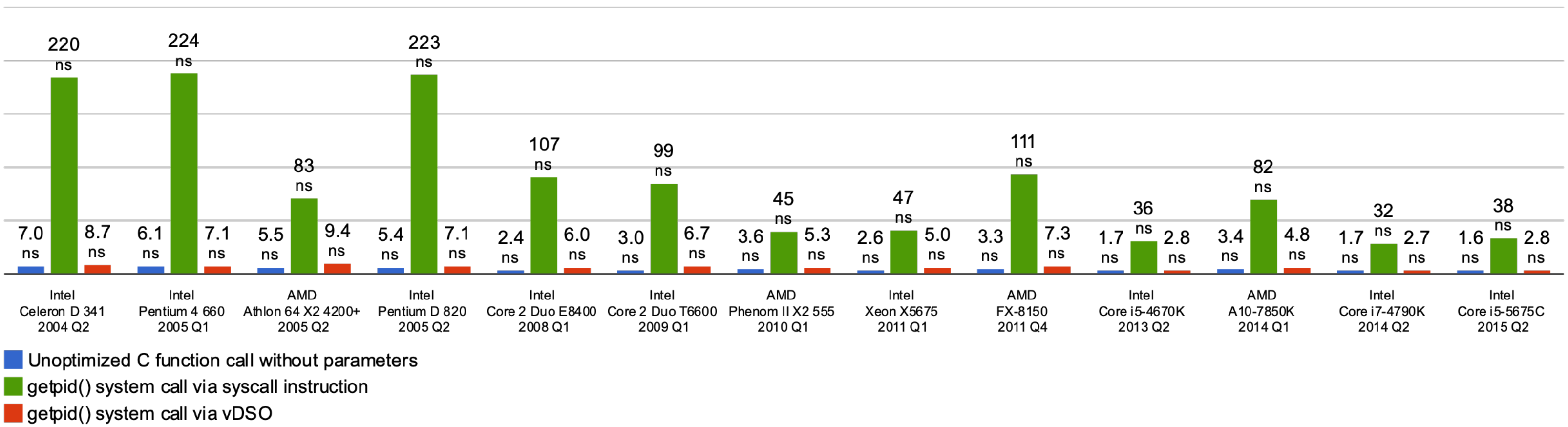
Uncontended Many-Lock Case



What if sys overhead is Y, even when the lock is free?

What if the OS can only handle one lock operation at a time?

Basic cost of a system call



- Min System call ~ 25x cost of function call
- Scheduling could be many times more
- Streamline system processing as much as possible
- Other optimizations seek to process as much of the call in user space as possible (eg, Linux vDSO)

A Better Lock Implementation

- Interrupt-based solution works for single core, but costly
- Doesn't work well on multi-core machines
  - Disable intr on all cores?

- Solution: Utilize hardware support for **atomic operations**

Recall: Atomic Operations

- Definition: **An operation runs to completion or not at all**
- Foundation for synchronization primitives

- Example: Loading or storing a word

Atomic Read-Modify-Write Instructions

- Problems with previous solution:
  - Works only in system Privilege level, not User level
  - Doesn't work well on multiprocessor
    - Disabling interrupts on all processors time consuming and undermines HW parallelism
- Alternative: atomic instruction sequences
  - These instructions read a value and write a new value atomically
  - Hardware is responsible for implementing this correctly
    - on both uniprocessors (not too hard)
    - and multiprocessors (requires help from cache coherence protocol)
  - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors
  - Natural extensions to user-level locking

Examples of Read-Modify-Write

- ```
  test&set (&address) {          /* most architectures */
       result = M[address];      // return result from "address" and
       M[address] = 1;           // set value at "address" to 1
       return result;
  }
  ```
  as if it happened all at once

- ```
  swap (&address, register) {    /* x86 */
       temp = M[address];        // swap register's value to
       M[address] = register;    // value at "address"
       register = temp;
  }
  ```

- ```
  compare&swap (&address, reg1, reg2) { /* 68000 */
       if (reg1 == M[address]) {  // If memory still == reg1,
            M[address] = reg2;    // then  put reg2 => memory
            return success;
       } else {                   // Otherwise do not change memory
            return failure;
       }
  }
  ```

- ```
  load-linked&store-conditional(&address) { /* R4000, alpha */
       loop:
            ll r1, M[address];
            movi r2, 1;           // Can do arbitrary computation
            sc r2, M[address];
            beqz r2, loop;
  }
  ```

Implementing Locks with test&set

- Simple, but flawed, solution:

```
int value = 0; // Free

Acquire() {
   while (test&set(value)) {}; // spin while busy
}

Release() {
   value = 0;                  // atomic store
}
```

- Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
  - If lock is busy, test&set reads 1 and sets value=1 (no change) It returns 1, so while loop continues.
  - When we set value = 0, someone else can get lock.
- Busy-Waiting: thread consumes cycles while waiting
  - For multiprocessors: every test&set() is a write, which makes value ping-pong around in cache (using lots of memory BW)

Problem: Busy-Waiting for Lock

- Positives for this solution
  - Machine can receive interrupts
  - User code can use this lock (poorly)
  - Works on a multiprocessor
- Negatives
  - This is very inefficient as thread will consume cycles waiting
  - Waiting thread may take cycles away from thread holding lock (no one wins!)
  - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock $\Rightarrow$ no progress!
- For semaphores (and monitors), waiting thread may wait for an arbitrary long time!
  - Thus even if busy-waiting was OK for locks, definitely not ok for other primitives
  - Homework/exam solutions should avoid busy-waiting!

Multiprocessor Spin Locks: test&test&set

- A better solution for multiprocessors:

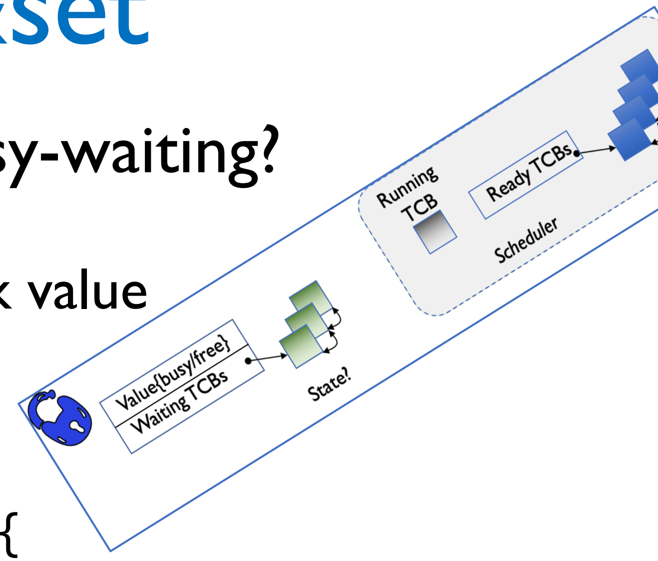
```
int mylock = 0; // Free
Acquire() {
   do {
      while(mylock);    // Wait until might be free
   } while(test&set(&mylock)); // exit if get lock
}

Release() {
   mylock = 0;
}
```

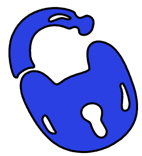
- Simple explanation:
  - Wait until lock might be free (only reading – stays in cache)
  - Then, try to grab lock with test&set
  - Repeat if fail to actually get lock
- Issues with this solution:
  - Busy-Waiting: thread still consumes cycles while waiting
    - However, it does not impact other processors!

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically *check* lock value




```
int guard = 0;
int value = FREE;
```



```
Acquire() {
   // Short busy-wait time
   while (test&set(guard));
   if (value == BUSY) {
      put thread on wait queue;
      go to sleep() & guard = 0;
   } else {
      value = BUSY;
      guard = 0;
   }
}
```

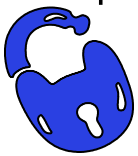
```
Release() {
   // Short busy-wait time
   while (test&set(guard));
   if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   guard = 0;
```

- Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?

Locks using Interrupts vs. test&set

Recall "disable interrupt" solution

```
int value = FREE;
```



```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

Basically we replaced:

- `disable interrupts` → `while (test&set(guard));`
- `enable interrupts` → `guard = 0;`

Recap: Locks using interrupts

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
}
```

```
Release() {
  enable interrupts;
}
```

If one thread in critical section, no other activity (including OS) can run!

```
int value = 0;
Acquire() {
  // Short busy-wait time
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}
```

```
Release() {
  // Short busy-wait time
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Recap: Locks using test & set

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
int value = 0;
Acquire() {
  while(test&set(value));
}

Release() {
  value = 0;
}
```

Threads waiting to enter critical section busy-wait

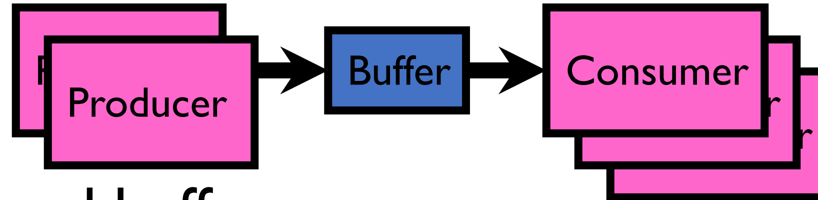
```
int guard = 0;
int value = 0;
Acquire() {
  // Short busy-wait time
  while(test&set(guard));
  if (value == 1) {
    put thread on wait-queue;
    go to sleep()& guard = 0;
  } else {
    value = 1;
    guard = 0;
  }
}

Release() {
  // Short busy-wait time
  while (test&set(guard));
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  guard = 0;
}
```

Higher-level Primitives

- Goal of last couple of lectures:
  - What is right abstraction for synchronizing threads that share memory?
  - Want as high a level primitive as possible
- Good primitives and practices important!
  - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs
- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  - Requires both mutual exclusion and cooperation (or orchestration)

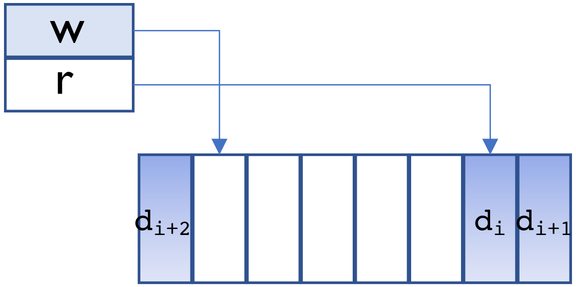
Producer-Consumer with a Bounded Buffer



- Problem Definition
  - Producers puts things into a shared buffer
  - Consumers takes them out
- Don't want producers and consumers to have to work in lockstep, so put a buffer (bounded) between them
  - Need synchronization to maintain integrity of the data structure and coordinate producers/consumers
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty
- GCC compiler – simple `1-1`
  - `cpp | cc1 | cc2 | as | ld`
- `Web servers,Routers,` …

Circular Buffer Data Structure (sequential case)

```
typedef struct buf {
  int write_index;
  int read_index;
  <type> *entries[BUFSIZE];
} buf_t;
```



- Insert: write & bump write ptr (enqueue)
- Remove: read & bump read ptr (dequeue)
- *How to tell if Full (on insert) Empty (on remove)?*
- *And what do you do if it is?*
- *What needs to be atomic?*

Producer/Consumer Correctness

- With multiple threads, each waits for the other to make process
- Scheduling constraints:
  - Consumer waits for producer if buffer is empty
  - Producer waits for consumer if buffer is full
- Mutual Exclusion: Only one thread manipulates the buffer data structure at a time

Lock Solution – first cut

```
mutex buf_lock = <initially unlocked>
```

```
Producer(item) {
  lock buffer
  while (buffer full) {}; // Wait for a free slot
  Enqueue(item);
  unlock buffer
}
```

Will we ever come out of the wait loop?

```
Consumer() {
  lock buffer
  while (buffer empty) {}; // Wait for arrival
  item = queue();
  unlock buffer
  return item
}
```

Lock Solution – 2nd cut

```
mutex buf_lock = <initially unlocked>
```

```
Producer(item) {
  lock buffer
  while (buffer full) {unlock; lock;};
  Enqueue(item);
  unlock buffer
}
```

```
Consumer() {
  lock buffer
  while (buffer empty) {unlock; lock;};
  item = queue();
  unlock buffer
  return item
}
```

What happens when one is waiting for the other?
- Multiple cores ?
- Single core ?

Explore semaphore solution

- **One semaphore per constraint**
  1. Mutex (mutual exclusion)
  2. Filled Slots (consumer waits if necessary)
  3. Empty Slots (producer waits if necessary)

Producer/Consumer Code

```
Semaphore fullSlots = 0; // Buffer empty to start
Semaphore emptySlots = bufSize; // All slots empty
Semaphore mutex = 1; // No one in critical sect.

 Producer(item) {
   emptySlots.P(); // Wait for a free slot
   mutex.P();      // down
   Enqueue(item);
   mutex.V();      // up
   fullSlots.V(); // Tell consumers about new data
 }
```

Producer/Consumer Code

```
Semaphore fullSlots = 0; // Queue empty to start
Semaphore emptySlots = bufSize; // All slots empty
Semaphore mutex = 1; // No one in critical sect.
```

```
Consumer() {
  fullSlots.P(); // Wait for an item to be present
  mutex.P();
  item = Dequeue();
  mutex.V();
  emptySlots.V(); // Tell producers about new slot
  return item;
}
```

Producer/Consumer Code

```
Semaphore fullSlots = 0; // Queue empty to start
Semaphore emptySlots = bufSize; // All slots empty
Semaphore mutex = 1; // No one in critical sect.
```

```
Producer(item) {
  emptySlots.P();
  mutex.P();
  Enqueue(item);
  mutex.V();
  fullSlots.V();
}
```

```
Consumer() {
  fullSlots.P();
  mutex.P();
  item = Dequeue();
  mutex.V();
  emptySlots.V();
  return item;
}
```

Discussion

- **What if we wrote the following?**

```
Producer(item) {
  mutex.P();
  emptySlots.P();
  Enqueue(item);
  mutex.V();
  fullSlots.V();
}
```

```
Consumer() {
  fullSlots.P();
  mutex.P();
  item = Dequeue();
  mutex.V();
  emptySlots.V();
  return item;
}
```

Deadlock … More on this later

Discussion

- **What about this?**

```
Producer(item) {
  emptySlots.P();
  mutex.P();
  Enqueue(item);
  fullSlots.V();
  mutex.V();
}
```

```
Consumer() {
  fullSlots.P();
  mutex.P();
  item = Dequeue();
  mutex.V();
  emptySlots.V();
  return item;
}
```

Correct, possibly less efficient

Problems with Semaphores

- More powerful (and primitive) than locks
- Argument: Clearer to have separate constructs for
  - Mutual Exclusion: One thread can do something at a time
  - Waiting for a condition to become true
- Need to make sure a thread calls P() for every V()
  - Other tools are more flexible than this

Break

Condition Variables

- **Collection of threads waiting *inside* a critical section**
- Operations:
  - **`wait(&lock)`:** Atomically release lock and go to sleep. Re-acquire the lock before returning.
  - **`signal()`:** Wake up on waiting thread (if there is one)
  - **`broadcast()`**: Wake up all waiting threads

- **Rule:** Hold lock when using a condition variable

Lock Solution – 2nd cut

```
mutex buf_lock = <initially unlocked>
Condvar buf_signal = <initially nobody>
```

```
Producer(item) {
  lock buffer
  while (buffer full) {cond_wait(buf_signal, buf_lock) };
  Enqueue(item);
  unlock buffer
}
```

```
Consumer() {
  lock buffer
  while (buffer empty) {cond_wait(buf_signal, buf_lock) };
  item = queue();
  unlock buffer
  return item
}
```

Release lock; signal others to run; reacquire on resume

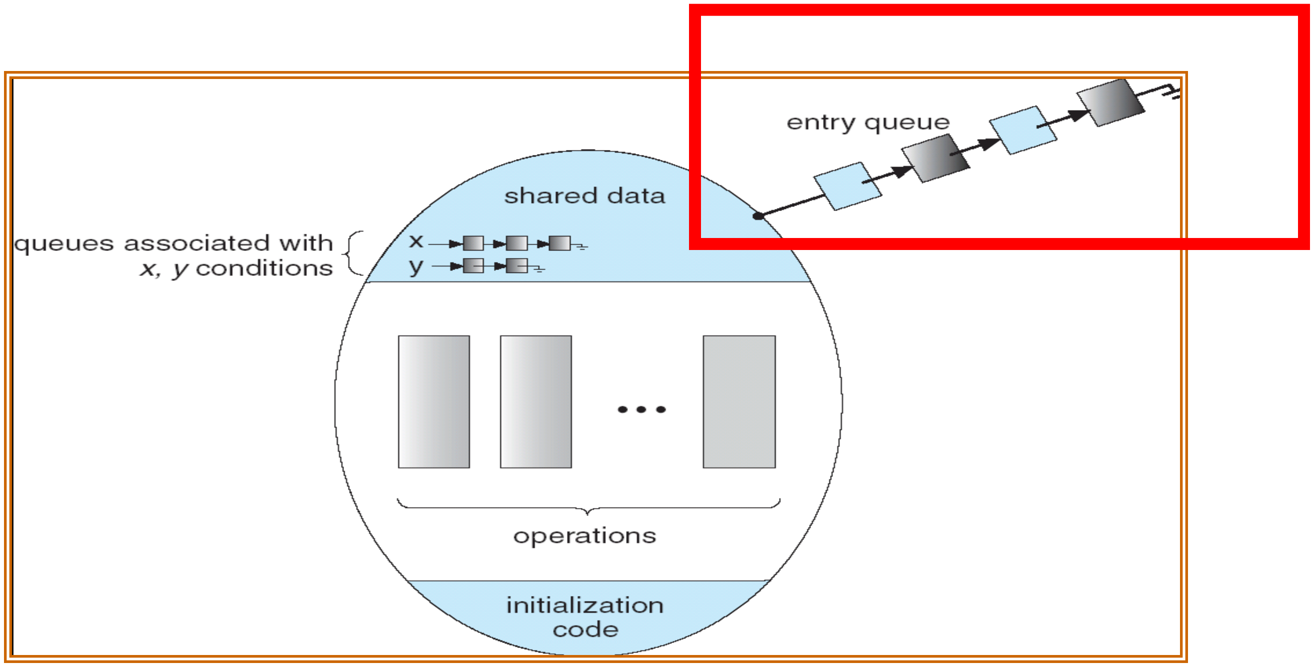
n.b. OS must do the reacquire

Why User must recheck?

Why the `while` Loop?

- When a thread is woken up by `signal()`, it is simply put on the ready queue
- It may or may not reacquire the lock immediately!
  - Another thread could be scheduled first and "sneak in" to empty the queue
  - Need a loop to re-check condition on wakeup

Monitors

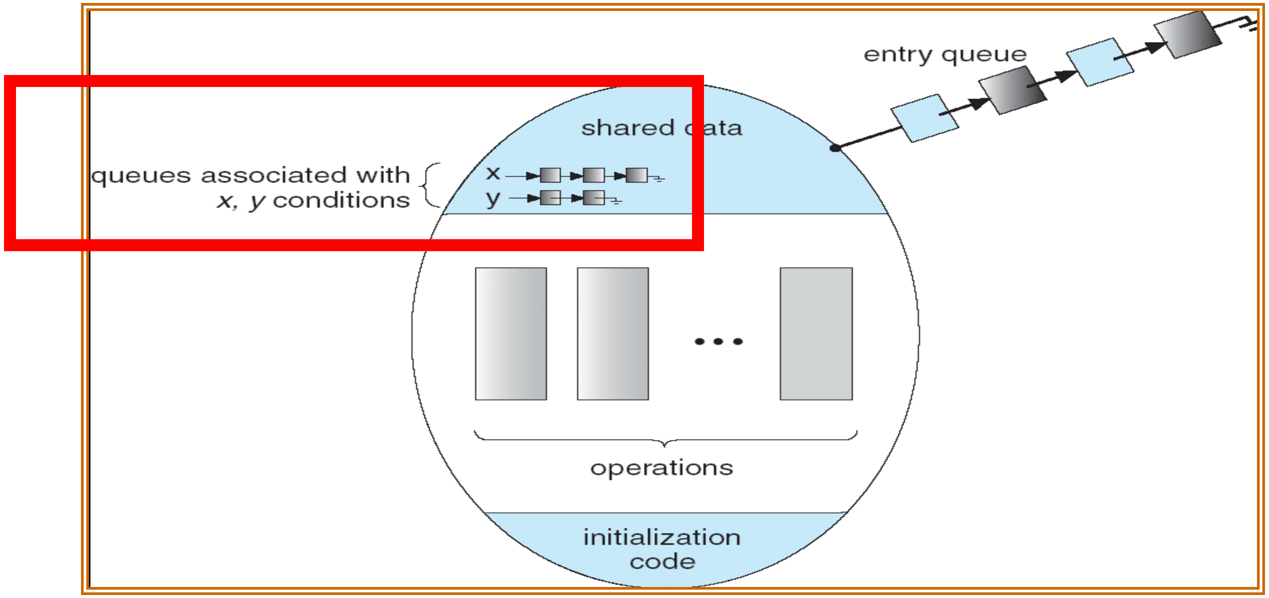


- **Lock:** protects access to shared data
- Always acquire lock when accessing
- Queue of threads waiting to enter the monitor

Monitors in practice

- Locks for mutual exclusion
- Condition variables for waiting
- A **monitor** is a lock and zero or more condition variables with some associated data and operations
  - Java provides this natively
  - POSIX threads: Provides **locks** and **condvars**, have to build your own

Monitors



- **Condition Variables:** queue of threads waiting for something to become true inside critical sect.
- Atomically release lock and start waiting
  - Another thread using the monitor will signal them
- The condition: Some function of monitor's data

Why the `while` Loop?

- Can we "hand off" the lock directly to the signaled thread so no other thread "sneaks in?"
  - Yes. Called **Hoare-Style Monitors**
  - Many textbooks describe this scheme
- Most OSs implement **Mesa-Style Monitors**
  - Allows other threads to sneak in
  - Much easier to implement
  - Even easier if you allow "spurious wakeups"
  - `wait()` can return when no signal occurred, in rare cases
  - POSIX allows spurious wakeups

Interlude: Concurrency Is Hard

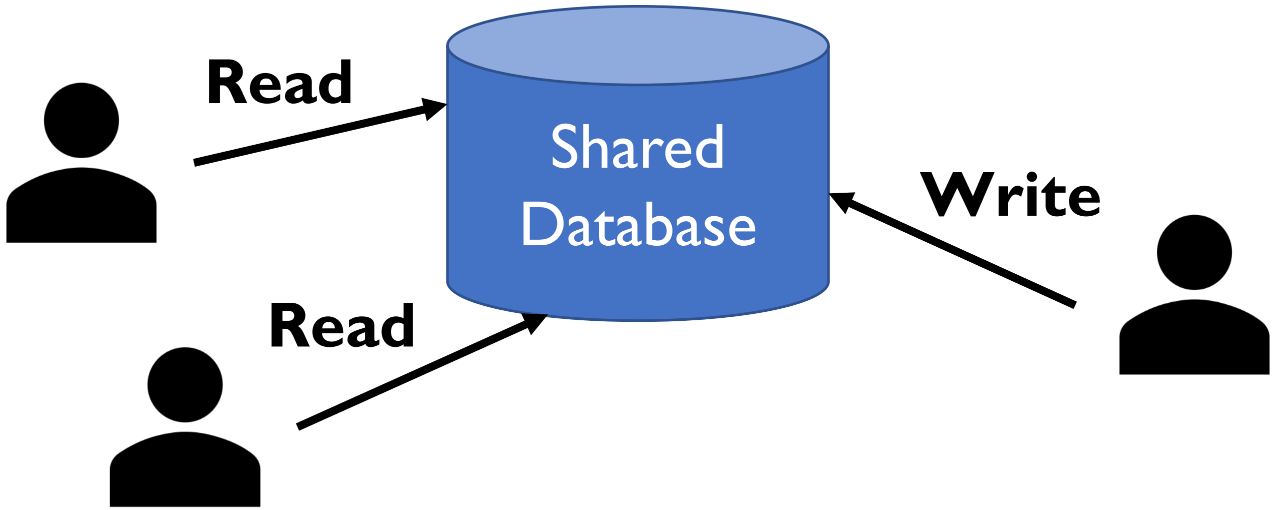
- Even for practicing engineers trying to write mission-critical, bulletproof code!
- Therac-25: Radiation Therapy Machine with Unintended Overdoses (reading on course site)
- Mars Pathfinder Priority Inversion (JPL Account)
- Toyota Uncontrolled Acceleration (CMU Talk)
    - 256.6K Lines of C Code, ~9-11K global variables
    - Inconsistent mutual exclusion on reads/writes

Comparing Synchronization

- Semaphores can implement locks
  - `Acquire() { semaphore.P(); }`
  - `Release() { semaphore.V(); }`
- and Condition Variables
- Monitors combine locks and CVs in a structured fashion
- Modern view: concurrent objects (e.g., Java)
- Can monitors implement semaphores?
- Are there other important common patterns?

Time Permitting …

Reader/Writer Problem



- Shared Database
  - Many readers – never modify the database
  - Few writers – read and modify database
- Single lock sufficient?

Reader/Writer Correctness

- Readers can access when no writers
- Writers can access when no readers **and no other writers**

- A lock will satisfy these requirements
  - But we want to allow **multiple readers**
  - Better efficiency

Reader/Writer with Monitors

```
Reader() {
  Wait until no active writers
  Access database
  Maybe wake up a writer
}
Writer() {
  Wait until no active readers or writers
  Access database
  Maybe wakeup reader or writer
}
```

Lock (for mutual exclusion)

int activeReaders, condVar okToRead

int activeWriters, condVar okToWrite

Reader Version 1

```
Reader() {
    // First check self into system
    lock.Acquire();
    while (AW > 0) { // Is it safe to read?
       okToRead.wait(&lock); // Sleep on cond var
    }
    AR++;  // Now we are active!
    lock.release();
    // Perform actual read-only access
    AccessDatabase(ReadOnly);
    // Now, check out of system
    lock.Acquire();
    AR--;    // No longer active
    if (AR == 0)  // No other active readers
       okToWrite.signal();  // Wake up one writer
    lock.Release();
 }
```

Writer Version 1

```
Writer() {
    // First check self into system
    lock.Acquire();
    while (AR > 0 || AW > 0) { // Is it safe to write?
        okToWrite.wait(&lock); // Sleep on cond var
    }
    AW++;  // Now we are active!
    lock.release();
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    lock.Acquire();
    AW--;     // No longer active
    okToWrite.signal(); // Wake up one writer
    okToRead.broadcast(); // Wake up all readers
    lock.Release();
}
```

Writer Version 1: Starvation

```
Writer() {
    // First check self into system
    lock.Acquire();
    while (AR > 0 || AW > 0) { //
        okToWrite.wait(&lock); //
    }
    AW++;  // Now we are active!
    lock.release();
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    lock.Acquire();
    AW--;     // No longer active
    okToWrite.signal(); // Wake up one writer
    okToRead.broadcast(); // Wake up all readers
    lock.Release();
}
```

If there are always readers, this is always true! Writer starves

Writer Version 1: Conflict

```
Writer() {
    // First check self into system
    lock.Acquire();
    while (AR > 0 || AW > 0) { // Is it safe to write?
        okToWrite.wait(&lock); // Sleep on cond var
    }
    AW++;  // Now we are active!
    lock.release();
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    lock.Acquire();
    AW--;     // No longer active
    okToWrite.signal(); // Wake
    okToRead.broadcast(); // Wak
    lock.Release();
}
```

Relies on waiting threads double-checking condition

Writer Version 1: Conflict

```
Writer() {
    // First check self into system
    lock.Acquire();
    while (AR > 0 || AW > 0) { // Is it safe to write?
        okToWrite.wait(&lock); // Sleep on cond var
    }
    AW++;  // Now we are active!
    lock.release();
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    lock.Acquire();
    AW--;    // No longer activ
    okToWrite.signal(); // Wake
    okToRead.broadcast(); // Wak
    lock.Release();
}
```

Everyone races, all but 1 thread just goes back to sleep

Reader/Writer with Monitors v2

```
Reader() {
  Wait until no active or waiting writers
  Access database
  Maybe wake up a writer
}
Writer() {
  Wait until no active readers or writers
  Access database
  If waiting writer, wake it up;
  Otherwise, wakeup readers;
}
```

int waitingWriters

Reader Version 2

```
Reader() {
    // First check self into system
    lock.Acquire();
    while (AW > 0 || WW > 0) { // Is it safe to read?
       okToRead.wait(&lock); // Sleep on cond var
    }
    AR++;  // Now we are active!
    lock.release();
    // Perform actual read-only access
    AccessDatabase(ReadOnly);
    // Now, check out of system
    lock.Acquire();
    AR--;    // No longer active
    if (AR == 0)  // No other active readers
       okToWrite.signal();  // Wake up one writer
    lock.Release();
 }
```

Writer Version 2

```
Writer() {
    // First check self into system
    lock.Acquire();
    while (AR > 0 || AW > 0) { // Is it safe to write?
      WW++;
      okToWrite.wait(&lock); // Sleep on cond var
      WW--;
    }
    AW++;  // Now we are active!
    lock.release();
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    lock.Acquire();
    AW--;    // No longer active
    if (WW > 0)
      okToWrite.signal(); // Wake up one writer
    else
      okToRead.broadcast(); // Wake up all readers
    lock.Release();
}
```

Simulation of Reader/Writer

- Sequence of arrivals: R1, R2, W1, R3
- On entry each reader checks

```
while (AW > 0 || WW > 0) { // Is it safe to read?
      okToRead.wait(&lock); // Sleep on cond var
}
```

- First R1 enters (no waiting)
  - **AR = 1**, AW = 0, WW = 0
- Then R2 enters (no waiting)
  - **AR = 2**, AW = 0, WW = 0

Simulation of Reader/Writer

- Sequence of arrivals: R1, R2, *W1, R3
- **R1, R2 still running (AR = 2)**
- W1 does a check: AR > 0, waits on **`okToWrite`**

```
while (AR > 0 || AW > 0) { // Is it safe to write?
  WW++;
  okToWrite.wait(&lock); // Sleep on cond var
  WW--;
}
```

- Now AR = 2, AW = 0, **WW = 1**
- R3: **WW > 0**, waits on okToRead

Simulation of Reader/Writer

- R1 finishes, does not wake anyone up
  - AR = 1, AW = 0, WW = 1
- R2 finishes
  - AR = 0, AW = 0, WW = 1
  - Wakes up W1 (signals **`okToWrite`**)
- W1 runs and finishes
  - AR = 1, AW = 1 then 0, WW = 0
  - Wakes up R3 (**`okToRead.Broadcast()`**)

Reader/Writer Design Choices

- Reader starvation:

```
while (AW > 0 || WW > 0) { // Safe to read?
  okToRead.wait(&lock); // Sleep on cond var
}
```

- "Writer-biased" Lock
  - Can favor readers by changing conditions on wait loops
  - Other possibilities, e.g. track readers waiting since before current writer started

Summary

- Scheduling and Synchronization are Deeply Interrelated
- Synchronization overhead is a critical performance factor
- User-level Threads can remove OS-switch cost of synchronization, but lose the connection with scheduler
  - With lots of cores, this matters less
- Disabling interrupts is brute-force way to implement synchronization operations.
  - Does not play well with multiple cores. Cannot be used at User Level
- Hardware atomic read-modify-write provides a better solution
- Must be constructed carefully – spin on simple read (test & test-and-set)
- Synchronization involves both Mutual Exclusion and Signaling
  - Locks for Mutex, Condition Variables for signaling (cooperation)
- Semaphores: More primitive & general than locks, but used in both ways
- Alternative: Monitors
  - One lock, zero or more condition variables
- Reader/Writer Synchronization
  - Treat readers differently from writers for efficiency