CS330: Operating Systems

Threads

What is a thread?

- Threads are (almost!) independent execution entities of a single process

What is a thread?

- Threads are (almost!) independent execution entities of a single process
- Threads of a single process can be scheduled different CPUs in a concurrent manner.

What is a thread?

- Threads are (almost!) independent execution entities of a single process
- Threads of a single process can be scheduled different CPUs in a concurrent manner. Therefore,
  - Each thread has a different register state and stack
  - At a given point of time, PC of different threads can be different

What is a thread?

- Threads are (almost!) independent execution entities of a single process
- Threads of a single process can be scheduled different CPUs in a concurrent manner. Therefore,
  - Each thread has a different register state and stack
  - At a given point of time, PC of different threads can be different
- How threads are different from processes?

What is a thread?

- Threads are (almost!) independent execution entities of a single process
- Threads of a single process can be scheduled different CPUs in a concurrent manner. Therefore,
  - Each thread has a different register state and stack
  - At a given point of time, PC of different threads can be different
- How threads are different from processes?
  - Threads of a single process share the address space
  - Context switch between two threads of a process does not require switching the address space

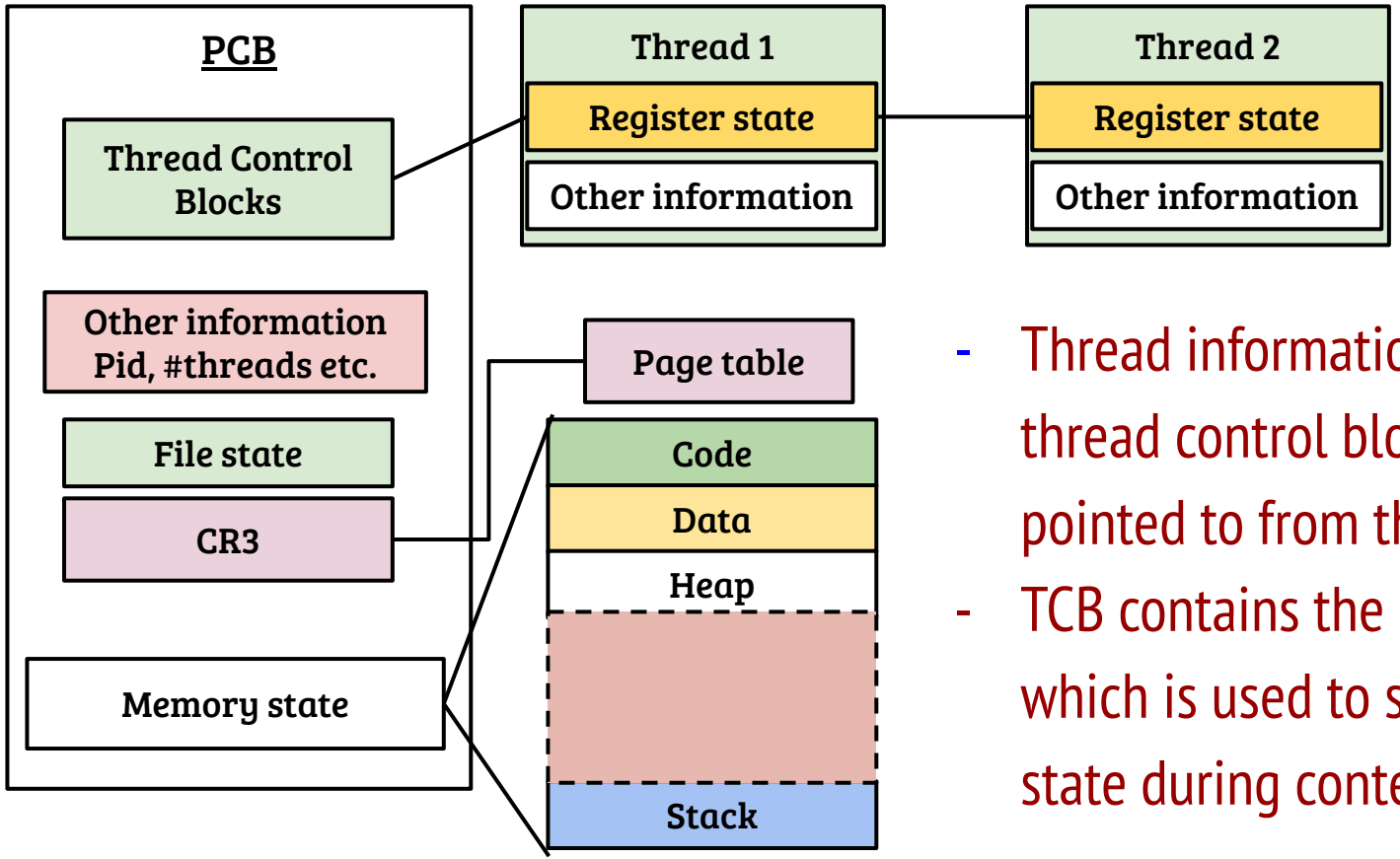
Multi-threaded processes

- Threads are (almost!) independent execution entities of a single process
- Why multithreading is useful?
- How does OS maintain thread related information?
- How stacks for multiple threads are managed?
- What is POSIX thread API? How is it used?
- How threads are different from processes?
  - Threads of a single process share the address space
  - Context switch between two threads of a process does not require switching the address space

Multi-threaded processes

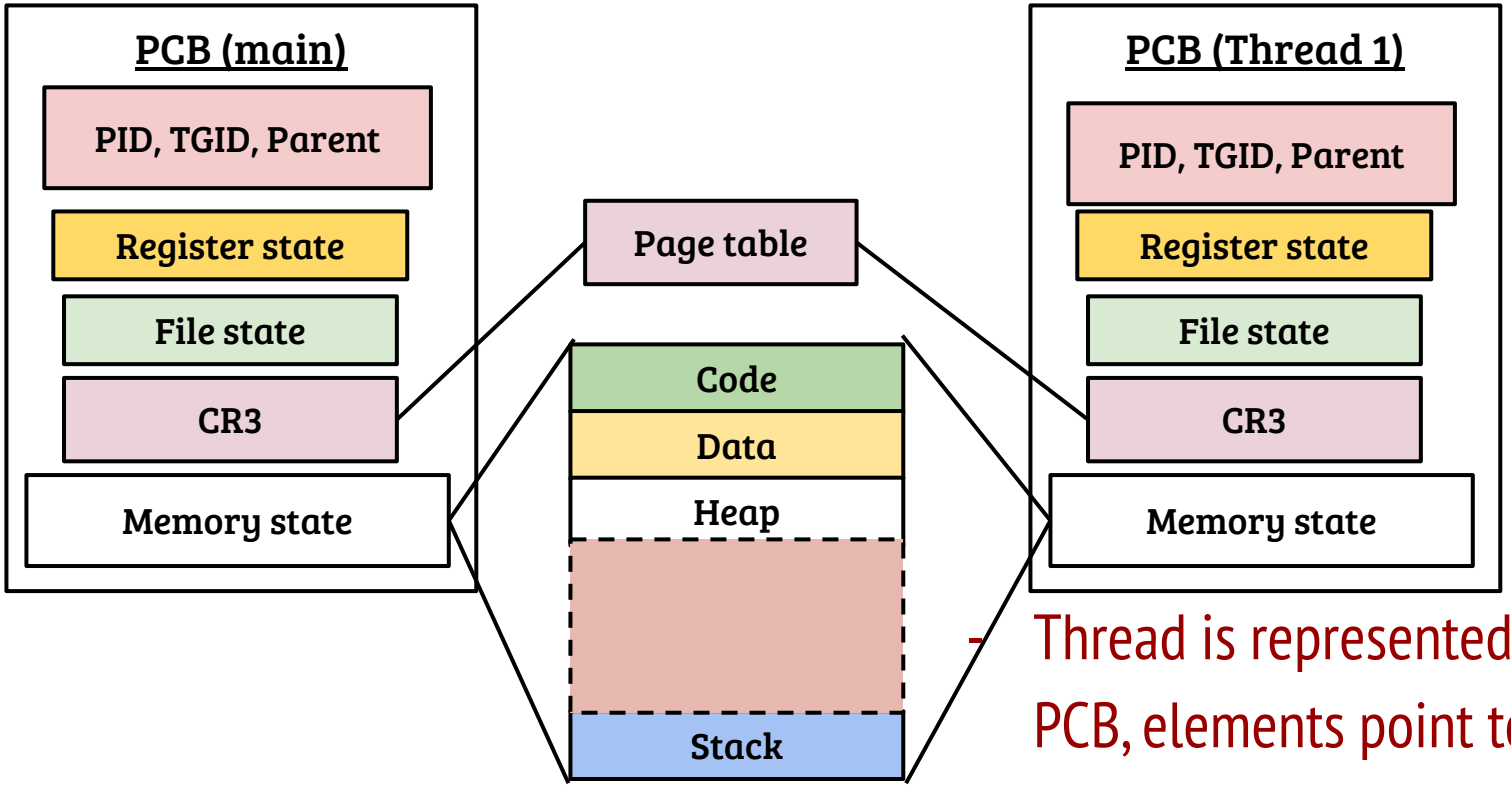
- Why multithreading is useful?
- Efficient execution on multicore systems, overlapping I/O and processing
- How does OS maintain thread related information?
- How stacks for multiple threads are managed?
- What is POSIX thread API? How is it used?

PCB of a multithreaded process



- Thread information is stored in thread control blocks (TCB) which is pointed to from the PCB
- TCB contains the register state which is used to save/restore CPU state during context switch

PCB of a multithreaded process (Linux)

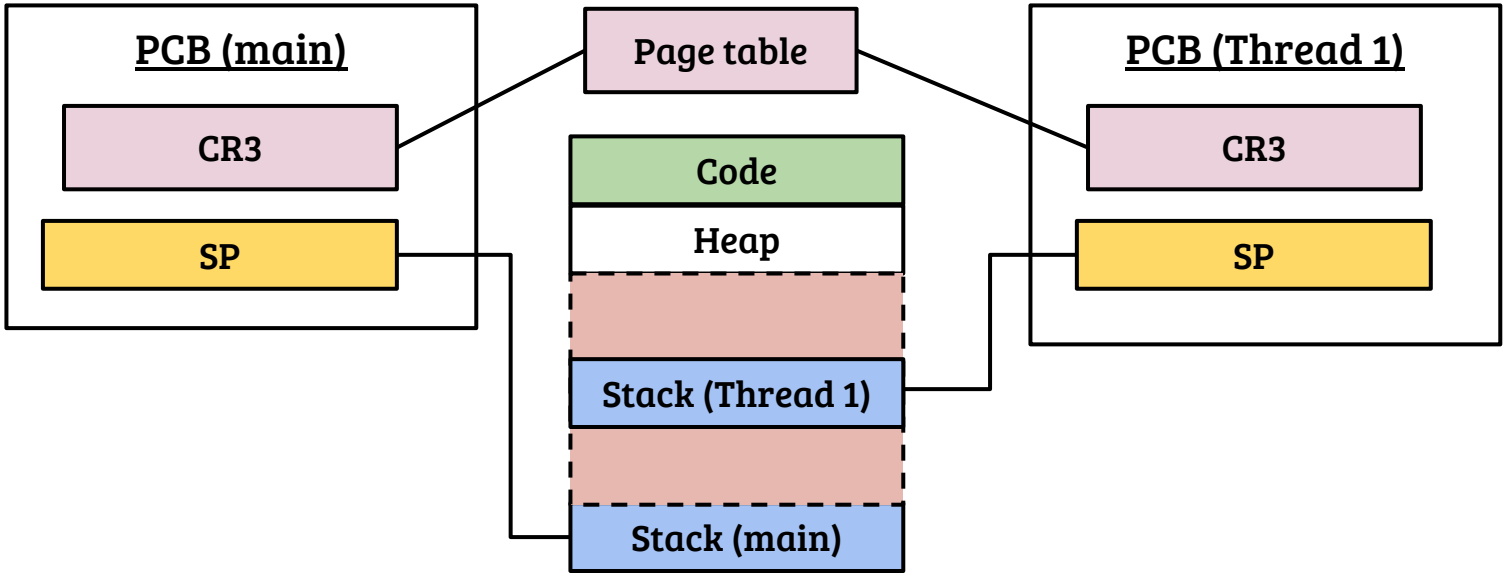


- Thread is represented by a separate PCB, elements point to the structure containing subsystem level info.

Multi-threaded processes

- Why multithreading is useful?
- Efficient execution on multicore systems, overlapping I/O and processing
- How does OS maintain thread related information?
- Maintain thread information using separate PCB or using TCB
- How stacks for multiple threads are managed?
- What is POSIX thread API? How is it used?

Stack for multi-threaded processes



- Stack for threads dynamically allocated from the address space using mmap( ) system call and passed to the OS during thread creation

Multi-threaded processes

Threads are (almost!) independent execution entities of a single process

- Why multithreading is useful?
- Efficient execution on multicore systems, overlapping I/O and processing
- How does OS maintain thread related information?
- Maintain thread information using separate PCB or using TCB
- How stacks for multiple threads are managed?
- Stacks for threads are allocated using memory allocation APIs
- What is POSIX thread API? How is it used?

switching the address space

Posix thread API (pthread_create)

```
int pthread_create( pthead_t *tid, pthread_attr_t *attr,
                    void * (*thfunc) (void*), void *arg);
```

- Creates a thread with "tid" as its handle and the thread starts executing the function pointed to by the "thfunc" argument
- A single argument (of type void *) can be passed to the thread
- Thread attribute can be used to control the thread behavior e.g., stack size, stack address etc. Passing NULL sets the defaults
- Returns 0 on success.
- Thread termination: return from thfunc, pthread_exit( ) or pthread_cancel( )
- In Linux, pthread_create and fork implemented using clone( ) system call

Posix thread API (pthread_join)

int pthread_join(pthead_t tid, void **retval)

- This call waits for the thread with handle “tid” to finish
- The return value of the thread is captured using the “retval” argument
  - The thread must allocate the return value which is freed after the process joins
- Invoking pthread_join for an already finished thread returns immediately

Multi-threaded processes

- Why multithreading is useful?
- Efficient execution on multicore systems, overlapping I/O and processing
- How does OS maintain thread related information?
- Maintain thread information using separate PCB or using TCB
- How stacks for multiple threads are managed?
- Stacks for threads are allocated using memory allocation APIs
- What is POSIX thread API? How is it used?
- Easy to use thread library with OS support. Important APIs: pthread_create, pthread_join

CS330: Operating Systems

Shared address space and concurrency

Threads sharing the address space is useful

- Threads share the address space
  - Global variables can be accessed from thread functions
  - Dynamically allocated memory can be passed as thread arguments
- Sharing data is convenient to design parallel computation

Threads sharing the address space is useful

- Threads share the address space
  - Global variables can be accessed from thread functions
  - Dynamically allocated memory can be passed as thread arguments
- Sharing data is convenient to design parallel computation
- Example parallel computation models
  - Data parallel processing: Data is partitioned into disjoint sets and assigned to different threads
  - Task parallel processing: Each thread performs a different computation on the same data

Example: Finding MAX

- Given *N* elements and a function *f*, we are required to find the element *e* such that *f(e)* is maximum
- If the computation time for function *f* is significant, we can employ multithreading with K threads using the following strategy
- Partition N elements into K non-overlapping sets and assign each thread to compute the MAX within its own set
- When all threads complete, we find out the global maximum

Threads sharing the address space

- Threads share the address space
  - Global variables can be accessed from thread functions

- Everything seems to be fine, what is the issue?
- How does OS fit into this discussion?

  - Data parallel processing: Data is partitioned into disjoint sets and assigned to different threads
  - Task parallel processing: Each thread performs a different computation on the same data

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
   int ctr = 0;
   for(ctr=0; ctr<100000; ++ctr)
         counter++;
}
```

- If this function is executed by two threads, what will be the value of *counter* when two threads complete?

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
   int ctr = 0;
   for(ctr=0; ctr<100000; ++ctr)
         counter++;
}
```

- If this function is executed by two threads, what will be the value of *counter* when two threads complete?
- Non-deterministic output
- Why?

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
   int ctr = 0;
   for(ctr=0; ctr<100000; ++ctr)
         counter++;
}
```

counter++ in assembly

```
mov (counter), R1
Add 1, R1
Mov R1, (counter)
```

Even on a single processor system, scheduling of threads between the above instructions can be problematic!

Sharing can be problematic!

T1: mov (counter), R1 // R1 = 0

T1: Add 1, R1

{switch-out, R1=1 saved in PCB}

- Assume that T1 is executing the first iteration
- On context switch, value of R1 is saved onto the PCB
- Thread T2 is scheduled and starts executing the loop

Sharing can be problematic!

T1: mov (counter), R1 // R1 = 0
T1: Add 1, R1
{switch-out, R1=1 saved in PCB}
T2: mov (counter), R1 // R1 = 0
T2: Add 1, R1 // R1 = 1
T2 mov R1, (counter) // counter = 1
{switch-out, T1 scheduled, R1 = 1}

- T2 executes all the instructions for one iteration of the loop, saves 1 to counter (in memory) and then, scheduled out
- T1 is switched-in, R1 value (=1) loaded from the PCB

Sharing can be problematic!

```
T1: mov (counter), R1    // R1 = 0
T1: Add 1, R1
{switch-out, R1=1 saved in PCB}
T2: mov (counter), R1    // R1 = 0
T2: Add 1, R1            // R1 = 1
T2 mov R1, (counter)     // counter = 1
{switch-out, T1 scheduled, R1 = 1}
T1: mov R1, (counter)    // counter = 1!
```

- T1 stores one into counter
- Value of counter should have been two
- What if "counter++" is compiled into a single instruction, e.g., "inc (counter)" ?

Sharing can be problematic!

```
T1: mov (counter), R1    // R1 = 0
T1: Add 1, R1
{switch-out, R1=1 saved in PCB}
T2: mov (counter), R1    // R1 = 0
T2: Add 1, R1            // R1 = 1
T2 mov R1, (counter)     // counter = 1
{switch-out, T1 scheduled, R1 = 1}
T1: mov R1, (counter)    // counter = 1!
```

- T1 stores one into counter
- Value of counter should have been two
- What if "counter++" is compiled into a single instruction, e.g., "inc (counter)" ?
- Does not solve the issue on multi-processor systems!

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
   int ctr = 0;
   for(ctr=0; ctr<100000; ++ctr)
         counter++;
}
```

- If this function is executed by two threads, what will be the value of *counter* when two threads complete?
- Non-deterministic output
- Why?
- Accessing shared variable in a concurrent manner results in incorrect output

Definitions

- Atomic operation: An operation is atomic if it is *uninterruptible* and *indivisible*
- Critical section: A section of code accessing one or more shared resource(s), mostly shared memory location(s)
- Mutual exclusion: Technique to allow exactly one execution entity to execute the critical section
- Lock: A mechanism used to orchestrate entry into critical section
- Race condition: Occurs when multiple threads are allowed to enter the critical section

Threads sharing the address space

- Threads share the address space
  - Global variables can be accessed from thread functions
- Everything seems to be fine, what is the issue?
- Correctness of program impacted because of concurrent access to the shared data causes race condition
- How does OS fit into this discussion?

assigned to different threads

  - Task parallel processing: Each thread performs a different computation on the same data

Critical sections in OS

- OS maintains shared information which can be accessed from different OS mode execution (e.g., system call handlers, interrupt handlers etc.)
- Example (1): Same page table entry being updated concurrently because of swapping (triggered because of low memory) and change of protection flags (because of mprotect( ) system call)
- Example (2): The queue of network packets being updated concurrently to deliver the packets to a process and receive incoming packets from the network device

Strategy to handle race conditions in OS

Contexts executing critical sections	Uniprocessor systems	Multiprocessor systems
System calls	Disable preemption	Locking
System calls, Interrupt handler	Disable interrupts	Locking + Interrupt disabling (local CPU)
Multiple interrupt handlers	Disable interrupts	Locking + Interrupt disabling (local CPU)

Threads sharing the address space

- Threads share the address space
- Everything seems to be fine, what is the issue?
- Correctness of program impacted because of concurrent access to the shared data causes race condition
- How does OS fit into this discussion?
- Concurrency issues in OS is challenging as finding the race condition itself is non-trivial

on the same data

CS330: Operating Systems

Locking

Recap: Strategy to handle race conditions in OS

Contexts executing critical sections	Uniprocessor systems	Multiprocessor systems
System calls	Disable preemption	Locking
System calls, Interrupt handler	Disable interrupts	Locking + Interrupt disabling (local CPU)
Multiple interrupt handlers	Disable interrupts	Locking + Interrupt disabling (local CPU)

Locking example: pthread mutex

```
pthread_mutex _t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
void *thfunc(void *)
{
  int ctr = 0;
  for(ctr=0; ctr<100000; ++ctr){
    pthread_mutex_lock(&lock);      // One thread acquires lock, others wait
    counter++;                      // Critical section
    pthread_mutex_unlock(&lock);  // Release the lock
  }
}
```

Design issues of locks

```
pthread_mutex _t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
    pthread_mutex_lock(&lock);     // One thread acquires lock, others wait
    counter++;                     // Critical section
    pthread_mutex_unlock(&lock);  // Release the lock
  }
}
```

- Efficiency of lock and unlock operations
- Lock acquisition delay vs. wasted CPU cycles
- Fairness of the locking scheme

Lock ADT

```
lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

```
lock_t *L1, L2;
....
lock(L1)
Critical Section
unlock(L1)
....
lock(L2)
Critical Section
unlock(L2)
....
Lock(L1)
Critical Section
unlock(L2)
```

Lock ADT: Efficiency

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

- Efficiency of lock/unlock operations directly influence performance
- Implementation choices?

Lock ADT: Efficiency

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

- Efficiency of lock/unlock operations directly influence performance
- Implementation choices?
- Hardware assisted implementations
  - Use hardware synchronization primitives like atomic operations

Lock ADT: Efficiency

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

- Efficiency of lock/unlock operations directly influence performance
- Implementation choices?
- Hardware assisted implementations
  - Use hardware synchronization primitives like atomic operations
- Software locks are implemented without assuming any hardware support
  - Not used in practice because of high overheads

Design issues of locks

```
pthread_mutex_t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
```

- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Fairness of the locking scheme

```
    counter++;                          // Critical section
    pthread_mutex_unlock(&lock);  // Release the lock
  }
}
```

Lock: busy-wait (spinlock) vs. Waiting

T1	T2
lock(L) //Acquired	
Critical section	lock(L) //Lock is busy. Reschedule or Spin?
unlock(L)	Critical section unlock(L)

Lock: busy-wait (spinlock) vs. Waiting

T1	T2
lock(L) //Acquired	
Critical section	lock(L) //Lock is busy. Reschedule or Spin?
unlock(L)	Critical section unlock(L)

- With busy waiting, context switch overheads saved, wasted CPU cycles due to spinning
- Busy waiting is prefered when critical section is small and the context executing the critical section is not rescheduled (e.g., due to I/O wait)

Design issues of locks

```
pthread_mutex_t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
```

- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Use waiting locks and spinlocks depending on the requirement
- Fairness of the locking scheme

```
    pthread_mutex_unlock(&lock); // Release the lock
  }
}
```

Fairness

- Given *N* threads contending for the lock, number of unsuccessful attempts for lock acquisition for all contending threads should be same
- Bounded wait property
  - Given *N* threads contending for the lock, there should be an upper bound on the number of attempts made by a given context to acquire the lock

Design issues of locks

- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Use waiting locks and spinlocks depending on the requirement
- Fairness of the locking scheme
- Contending threads should not starve for the lock

Spinlock: Buggy attempt

```
1.  lock_t *L = 0; // Initial value
2.  lock(L)
3.  {
4.    while(*lock);
5.    *lock = 1;
6.  }
7.  unlock(L)
8.  {
9.    *lock = 0;
10. }
```

- Does this implementation work?

Spinlock: Buggy attempt

```
1.  lock_t *L = 0; // Initial value
2.  lock(L)
3.  {
4.    while(*lock);
5.    *lock = 1;
6.  }
7.  unlock(L)
8.  {
9.    *lock = 0;
10. }
```

- Does this implementation work?
- No, it does not ensure *mutual exclusion*
- Why?

Spinlock: Buggy attempt

```
1.  lock_t *L = 0; // Initial value
2.  lock(L)
3.  {
4.    while(*lock);
5.    *lock = 1;
6.  }
7.  unlock(L)
8.  {
9.    *lock = 0;
10. }
```

- Does this implementation work?
- No, it does not ensure *mutual exclusion*
- Why?
  - Single core: Context switch between line #4 and line #5
  - Multicore: Two cores exiting the while loop by reading lock = 0

Spinlock: Buggy attempt

```
1.  lock_t *L = 0; // Initial value
2.  lock(L)
3.  {
4.    while(*lock);
5.    *lock = 1;
6.  }
7.  unlock(L)
8.  {
9.    *lock = 0;
10. }
```

- Does this implementation work?
- No, it does not ensure *mutual exclusion*
- Why?
  - Single core: Context switch between line #4 and line #5
  - Multicore: Two cores exiting the while loop by reading lock = 0
- Core issue: Compare and swap has to happen atomically!

Spinlock using atomic exchange

```
1. lock_t *L = 0; // Initial value
2. lock(L)
3. {
4.   while(atomic_xchg(*L, 1));
5. }
6. unlock(L)
7. {
8.   *lock = 0;
9. }
```

- Atomic exchange: exchange the value of memory and register atomically
- atomic_xchg (int *PTR, int val) returns the value at PTR before exchange
- Ensures mutual exclusion if “val” is stored on a register
- No fairness guarantees

Spinlock using XCHG on X86

```
lock(lock_t *L )
{
   asm volatile(
   “mov $1, %%rax;”
   “loop: xchg %%rax, (%%rdi);”
   “cmp $0, %%rax;”
    “jne loop;”
     : : : “memory” );
}
unlock(int *L ) { *L = 0;}
```

- XCHG *R*, *M* ⇒ Exchange value of register *R* and value at memory address *M*
- *RDI* register contains the lock argument
- Exercise: Visualize a context switch between any two instructions and analyse the correctness

Spinlock using compare and swap

```
1.  lock_t *L = 0; // Initial value
2.  lock(L)
3.  {
4.    while( CAS(*L, 0, 1) );
5.  }
6.  unlock(L)
7.  {
8.    *lock = 0;
9.  }
```

- Atomic compare and swap: perform the condition check and swap atomically
- CAS (int **PTR*, int *cmpval*, int *newval*) sets the value of *PTR* to *newval* if *cmpval* is equal to value at *PTR* . Returns 0 on successful exchange
- No fairness guarantees!

CAS on X86: cmpxchg

cmpxchg source[Reg] destination [Mem/Reg]
Implicit registers : rax and flags

```
1.  if rax == [destination]
2.  then
3.      flags[ZF] = 1
4.      [destination] = source
5.  else
6.      flags[ZF] = 0
7.      rax = [destination]
```

- "cmpxchg" is not atomic in X86, should be used with a "lock" prefix

Spinlock using CMPXCHG on X86

```
lock(lock_t *L )
{
asm volatile(
    “mov $1, %%rcx;”
    “loop: xor %%rax, %%rax;”
    “lock cmpxchg %%rcx, (%%rdi);”
    “jnz loop;”
    : : : “rcx”, “rax”, “memory”);
}
unlock(lock_t *L ) { *L = 0;}
```

- Value of RAX (=0) is compared against value at address in register RDI and exchanged with RCX (=1), if they are equal
- Exercise: Visualize a context switch between any two instructions and analyse the correctness

Load Linked (LL) and Store conditional (SC)

- LoadLinked (R, M)
  - Like a normal load, it loads R with value of M
  - Additionally, the hardware keeps track of future stores to M
- StoreConditional (R, M)
  - Stores the value of R to M if no stores happened to M after the execution of LL instruction (after execution, R = 1)
  - Otherwise, store is not performed (after execution R=0)
- Supported in RISC architectures like mips, risc-v etc.

Spinlock using LL and LC

```
lock_t *L = 0;
lock( lock_t *L)
{
 while(LoadLinked(L) ||
       !StoreConditional(L, 1));
}
unlock( lock_t *L) { *L = 0;}
```

```
lock:   LL   R1,  (R2);  //R2 = lock address
        BNEQZ  R1,  lock;
        ADDUI   R1,   R0, #1;    //R1 = 1
        SC R1, (R2)
        BEQZ R1, lock
```

- Efficient as the hardware avoids memory traffic for unsuccessful lock acquire attempts
- Context switch between LL and SC results in SC to fail

Spinlocks: reducing wasted cycles

- Spinning for locks can introduce significant CPU overheads and increase energy consumption
- How to reduce spinning in spinlocks?

Spinlocks: reducing wasted cycles

- Spinning for locks can introduce significant CPU overheads and increase energy consumption
- How to reduce spinning in spinlocks?
- Strategy: Back-off after every failure, exponential back-off used mostly

```
lock( lock_t *L) {
        u64 backoff = 0;
        while(LoadLinked(L) || !StoreConditional(L, 1)){
                if(backoff  < 63)   ++backoff;
                    pause(1 << backoff);   // Hint to processor
}
```

Fairness in spinlocks

- Spinlock implementations discussed so far are not fair,
  - no bounded waiting
- To ensure fairness, some notion of ordering is required
- What if the threads are granted the lock in the order of their arrival to the lock contention loop?
  - A single lock variable may not be sufficient
  - Example solution: Ticket spinlocks

Atomic fetch and add (xadd on X86)

xadd R, M

TmpReg T = R + [M]

R = [M]

[M] = T

- Example: M = 100; RAX = 200
- After executing “lock xadd %RAX, M”, value of RAX = 100, M = 300
- Require lock prefix to be atomic

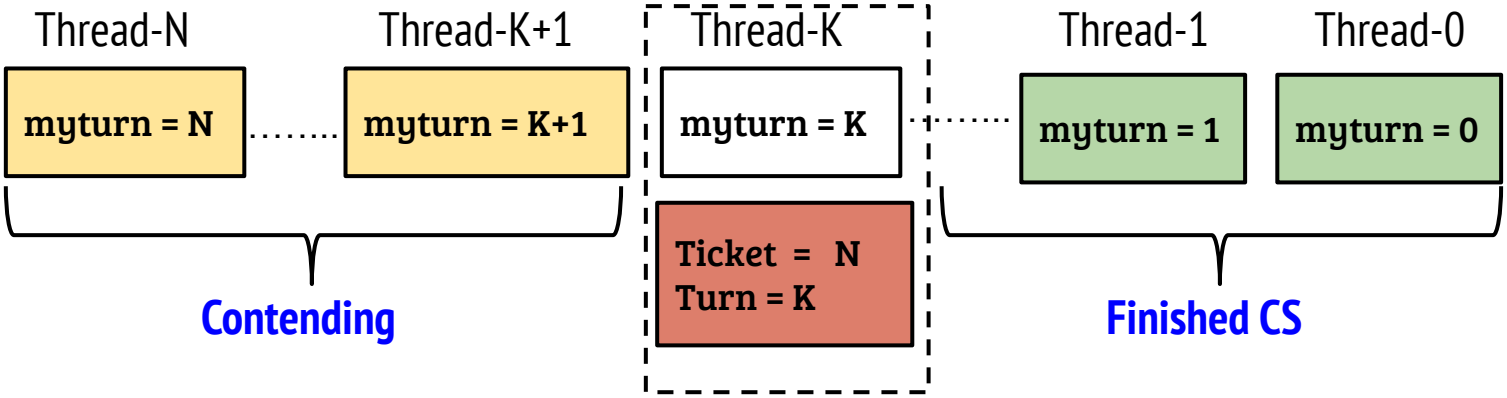
Ticket spinlocks (OSTEP Fig. 28.7)

```
struct lock_t{
        long ticket;
        long turn;
};
void init_lock (struct lock_t *L){
   L → ticket = 0;  L → turn = 0;
}
void unlock(struct lock_t *L){
      L → turn++;
}
```

```
void lock(struct lock_t *L){
   long myturn = xadd(&L → ticket, 1);
   while(myturn != L → turn)
        pause(myturn - L → turn);
}
```

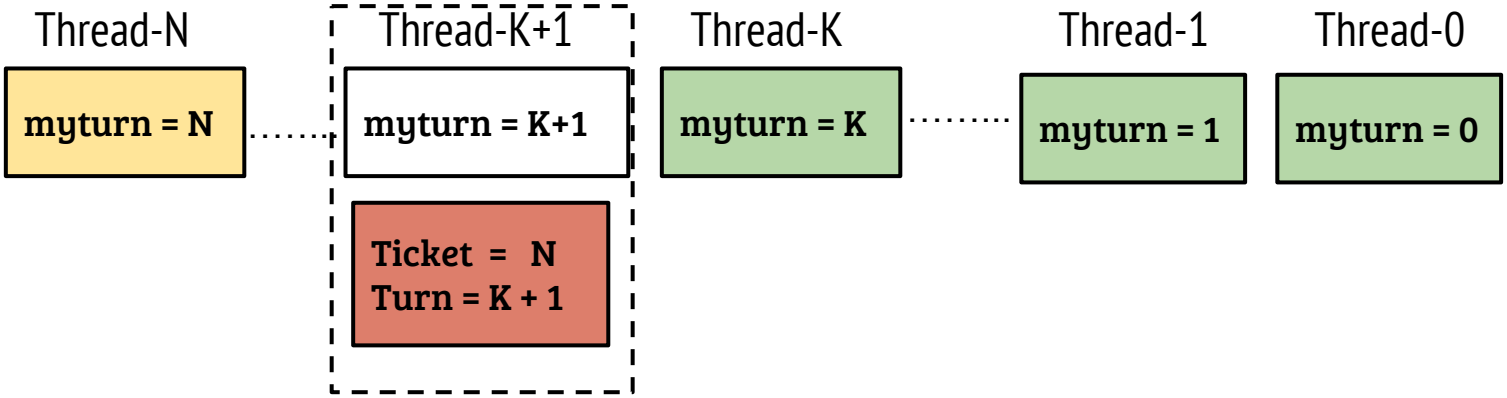
- Example: Order of arrival: T1 T2 T3
- T1 (in CS) : myturn = 0, L = {1, 0}
- T2: myturn = 1, L = {2, 0}
- T3: myturn = 2, L = {3,0}
- T1 unlocks, L = {3, 1}. T2 enters CS

Ticket spinlock



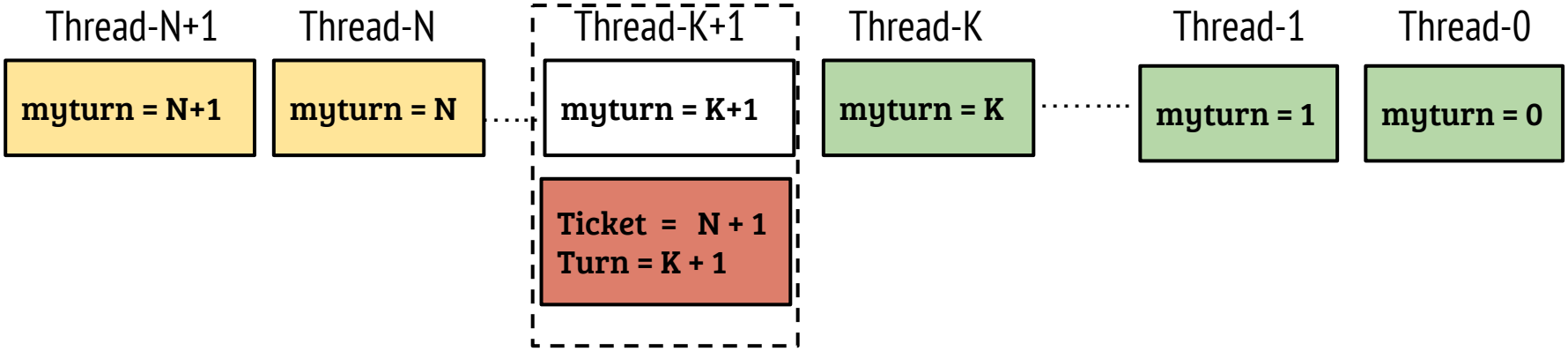
- Local variable “myturn” is equivalent to the order of arrival
- If a thread is in CS ⇒ Local Turn must be same as “Turn”
- Threads waiting = Ticket - Turn

Ticket spinlock



- Value of turn incremented on lock release
- Thread which arrived just after the current thread enters the CS
- When a new thread arrives, it gets the lock after the other threads ahead of the new thread acquire and release the lock

Ticket spinlock



- Ticket spinlock guarantees bounded waiting
- If *N* threads are contending for the lock and execution of the CS consumes *T* cycles, then bound = N * T (assuming negligible context switch overhead)

Ticket spinlock (with yield)

```
void lock(struct lock_t *L){
  long myturn = xadd(&L → ticket, 1);
  while(myturn != L → turn)
        sched_yield( );
}
```

- Why spin if the thread's turn is yet to come?
- Yield the CPU and allow the thread with ticket (or other non contending threads)
- Further optimization
  - Allow the thread with "myturn" value one less than "L⟶ turn" to continue spining

CS330: Operating Systems

Software locks, Semaphore

Recap: Spinlocks with hardware support

- Architectural support for atomic operations like atomic exchange, compare-and-swap, LL–SC and atomic add can be used to build spinlocks
- Ticket spinlocks provide fairness in locking, example implementation with atomic-add
- Outstanding issues: Blocking locks (will come back after semaphores)

Today's lecture: Software-only locks, Semaphores

Buggy #1

```
int flag[2] = {0,0};
void lock (int id)    /*id = 0 or 1 */
{
    while(flag[id ∧ 1]));  // ∧ → XOR
     flag[id] = 1;
}
void unlock (int id)
{
     flag[id] = 0;
}
```

- Solution for two threads, T_0 and T_1 with id 0 and 1, respectively
- We have seen that this solution does not work, Why?

Buggy #1

```
int flag[2] = {0,0};
void lock (int id)    /*id = 0 or 1 */
{
    while(flag[id ∧ 1]));  // ∧ → XOR
    flag[id] = 1;
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- Solution for two threads, T_0 and T_1 with id 0 and 1, respectively
- We have seen that this solution does not work, Why?
- Both threads can acquire the lock as "while condition check" and "setting the flag" is non-atomic

Buggy #2

```
int flag[2] = {0,0};
void lock (int id)    /*id = 0 or 1 */
{
    flag[id] = 1;
    while(flag[id ∧ 1]));  // ∧ → XOR
}
void unlock (int id)
{
     flag[id] = 0;
}
```

- Does this solution work?

Buggy #2

```
int flag[2] = {0,0};
void lock (int id)    /*id = 0 or 1 */
{
    flag[id] = 1;
    while(flag[id ∧ 1]));  // ∧ → XOR
}
void unlock (int id)
{
     flag[id] = 0;
}
```

- Does this solution work?
- No, as this can lead to a deadlock (flag[0] = flag[1] = 1) In other words the "progress" requirement is not met
- Progress: If no one has acquired the lock and there are contending threads, one of the threads must acquire the lock within a finite time

Buggy #3

```
int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   while(turn == id ∧ 1));
}
void unlock (int id)
{
    turn = id ∧ 1;
}
```

- Assuming T_0 invokes lock( ) first, does the solution provide mutual exclusion?

Buggy #3

```
int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   while(turn == id ∧ 1));
}
void unlock (int id)
{
    turn = id ∧ 1;
}
```

- Assuming T_0 invokes lock( ) first, does the solution provide mutual exclusion?
- Yes it does, but there is another issue with this solution - two threads must request the lock in an alternate manner
- Progress requirement is not met
  - Argument: one of the threads stuck in an infinite loop

Buggy #4

```
int flag[2] = {0,0};  int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   turn  = id ^1;
   flag[id] = 1;
   while(flag[id ^ 1]) && turn == (id ^1));
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- Why this solution does not work?

Buggy #4

```
int flag[2] = {0,0};  int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   turn  = id ^1;
   flag[id] = 1;
   while(flag[id ^ 1]) && turn == (id ^1));
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- Why this solution does not work?
- Mutual exclusion is not satisfied if T_0 context switched after setting the turn = 1 and T_1 acquires the lock (and sets turn = 0 in the process which allows T_0 to acquire the lock)

Attempt #5 (Peterson's solution)

```
int flag[2] = {0,0};  int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   flag[id] = 1;
   turn  = id ^1;
   while(flag[id ^ 1]) && turn == (id ^1));
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- Homework: Prove that mutual exclusion is guaranteed
- What about fairness?

Attempt #5 (Peterson's solution)

```
int flag[2] = {0,0};  int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   flag[id] = 1;
   turn  = id ^1;
   while(flag[id ^ 1]) && turn == (id ^1));
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- Homework: Prove that mutual exclusion is guaranteed
- What about fairness?
- The lock is fair because if two threads are contending, they acquire the lock in an alternate manner
- Extending the solution to N threads is possible

Semaphores

- Mutual exclusion techniques allows exactly one thread to access the critical section which can be restrictive
- Consider a scenario when a finite array of size N is accessed from a set of producer and consumer threads. In this case,
  - At most N concurrent producers are allowed if array is empty
  - At most N concurrent consumers are allowed if array is full
  - If we use mutual exclusion techniques, only one producer or consumer is allowed at any point of time

Operations on semaphore

```
struct semaphore{
                    int value;
                    spinlock_t *lock;
                    queue *waitQ;
                }sem_t;

// Operations

sem_init(sem_t *sem, int init_value);
sem_wait(sem_t *sem);
sem_post(sem_t *sem);
```

- Semaphores can be initialized by passing an initial value
- *sem_wait* waits (if required) till the value becomes +ve and returns after decrementing the value
- *sem_post* increments the value and wakes up a waiting context
- Other notations: P-V, down-up, wait-signal

Unix semaphores

```
#include <semaphore.h>

main( ){
    sem_t  s;
    int K = 5;
    sem_init(&s, 0, K);
    sem_wait(&s);
    sem_post(&s);
}
```

- Can be used to in a multi-threaded process or across multiple processes
- If second argument is 0, the semaphore can be used from multiple threads
- Semaphores initialized with value = 1 (third argument) is called a binary semaphore and can be used to implement locks
- Initialize: sem_init(s, 0, 1)
  lock:sem_wait(s), unlock: sem_post(s)

Semaphore usage example: wait for child

```
child( ){
      ...
     sem_post(s);
     exit(0);
}
int main (void ){
          sem_init(s, 0);
         if(fork( ) == 0)
               child( );
         sem_wait(s);
}
```

- Assume that the semaphore is accessible from multiple processes, value initialized to zero
- If parent is scheduled after the child creation, it waits till child finishes
- If child is scheduled and exits before parent, parent does not wait for the semaphore

Semaphore usage example: ordering

```
A=0; B=0;

Thread-0 {
    A = 1;
    printf("B = %d\n", B);
}

Thread-1 {
        B=1;
        printf("A = %d\n", A);
}
```

- What are the possible outputs?

Semaphore usage example: ordering

```
A=0; B=0;

Thread-0 {
    A = 1;
    printf("B = %d\n", B);
}

Thread-1 {
        B=1;
        printf("A = %d\n", A);
}
```

- What are the possible outputs?
- (A = 1, B= 1), (A = 1, B = 0), (A = 0, B=1)
- How to guarantee A = 1, B= 1?

Semaphore usage example: ordering

```
sem_init(s1, 0);
A=0; B=0;
Thread - 0 {
    A = 1;
    sem_wait(s1);
    printf("B = %d\n", B);
}
Thread - 1 {
        B=1;
        sem_post(s1);
        printf("A = %d\n", A);
}
```

- What are the possible outputs?

Semaphore usage example: ordering

```
sem_init(s1, 0);
A=0; B=0;
Thread - 0 {
    A = 1;
    sem_wait(s1);
    printf("B = %d\n", B);
}
Thread - 1 {
        B=1;
        sem_post(s1);
        printf("A = %d\n", A);
}
```

- What are the possible outputs?
- (A = 1, B = 1), (A=0, B=1)
- How to guarantee A = 1, B= 1?

Ordering with two semaphores

```
sem_init(s1, 0);
sem_init(s2, 0)
A=0; B=0;

Thread - 0
{
    A = 1;
    sem_post(s1);
    sem_wait(s2);
    printf("%d\n", B);
}
```

```
Thread - 1
{
    B=1;
    sem_wait(s1);
    sem_post(s2);
    printf("%d\n", A);
}
```

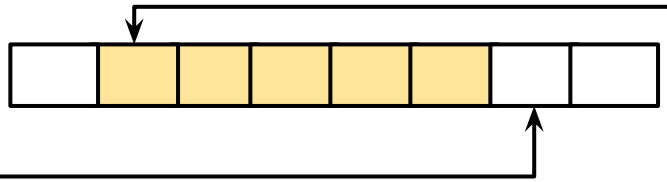
- Waiting for each other guarantees desired output

CS330: Operating Systems

Producer consumer problem

Producer-consumer problem

```
DoProducerWork( ){
 while(1){
   item_t item = prod_p( );
   produce(item);
 }
}
```



```
DoConsumerWork( ){
 while(1){
   item_t item = consume( );
   cons_p(item);
 }
}
```

- A buffer of size N, one or more producers and consumers
- Producer produces an element into the buffer (after processing)
- Consumer extracts an element from the buffer and processes it
- Example: A multithreaded web server, network protocol layers etc.
- Today's agenda: Solution using semaphores

Buggy #1

```
item_t A[n], pctr=0, cctr = 0;
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used);
    item_t item = A[cctr];
    cctr = (cctr + 1) % n;
    sem_post(&empty);
    return item;
}
```

- This solution does not work. What is the issue?

Buggy #1

```
item_t A[n], pctr=0, cctr = 0;
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used);
    item_t item = A[cctr];;
    cctr = (cctr + 1) % n;
    sem_post(&empty);
    return item;
}
```

- This solution does not work. What is the issue?
- The counters (pctr and cctr) are not protected, can cause race conditions

Buggy #2

```
item_t A[n], pctr=0, cctr = 0; lock_t *L = init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    lock(L);  sem_wait(&empty);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    sem_post(&used);  unlock(L);
}
```

```
item_t consume( ) {
    lock(L);  sem_wait(&used);
    item_t item = A[cctr];;
    cctr = (cctr + 1) % n;
    sem_post(&empty); unlock(L);
    return item;
}
```

- What is the problem?

Buggy #2

```
item_t A[n], pctr=0, cctr = 0; lock_t *L = init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    lock(L);  sem_wait(&empty);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    sem_post(&used);  unlock(L);
}
```

```
item_t consume( ) {
    lock(L);  sem_wait(&used);
    item_t  item = A[cctr];;
    cctr = (cctr + 1) % n;
    sem_post(&empty); unlock(L);
    return item;
}
```

- What is the problem?
- Consider empty = 0 and producer has taken lock before the consumer. This results in a deadlock, consumer waits for L and producer for empty

A working solution

```
item_t A[n], pctr=0, cctr = 0;  lock_t *L = init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty); lock(L);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    unlock(L); sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used); lock(L)
    item_t item = A[cctr];
    cctr = (cctr + 1) % n;
    unlock(L); sem_post(&empty);
    return item;
}
```

- The solution is deadlock free and ensures correct synchronization, but very much serialized (inside produce and consume)
- What if we use separate locks for producer and consumer?

Solution with separate mutexes

```
item_t A[n], pctr=0, cctr = 0;  lock_t *P = init_lock( ), *C=init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty); lock(P);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    unlock(P); sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used); lock(C)
    item_t item = A[cctr];
    cctr = (cctr + 1) % n;
    unlock(C); sem_post(&empty);
    return item;
}
```

- Does this solution work?
- Homework: Assume that item is a large object which has to be copied. How can we avoid copy of item while holding the lock?

Reader-writer locks

- Allows *multiple readers* or *a single writer* to enter the CS
- Example: Insert, delete and lookup operations on a search tree

Reader-writer locks

- Allows *multiple readers* or *a single writer* to enter the CS
- Example: Insert, delete and lookup operations on a search tree

```
struct BST{
        struct node *root;
        rwlock_t *lock;
};
```

```
struct node{
        item_t item;
        struct node *left;
        struct node*right;
};
```

```
void insert(BST *t, item_t item);
void lookup(BST *t, item_t item);
```

Reader-writer locks

- Allows *multiple readers* or *a single writer* to enter the CS
- Example: Insert, delete and lookup operations on a search tree

```
struct BST{
        struct node *root;
        rwlock_t *lock;
};
```

```
struct node{
        item_t item;
        struct node *left;
        struct node*right;
};
```

```
void insert(BST *t, item_t item);
void lookup(BST *t, item_t item);
```

- If multiple threads call lookup( ), they may traverse the tree in parallel

Implementation of read-write locks

```
struct rwlock_t{
    Lock read_lock;
    Lock write_lock;
    int num_readers;
}
```

```
init_lock(rwlock_t *rL)
{
   init_lock(&rL → read_lock);
   init_lock(&rL → write_lock);
   rL → num_readers = 0;
}
```

Implementation of read-write locks (writers)

```
struct rwlock_t{
    Lock read_lock;
    Lock write_lock;
    int num_readers;
}

void write_lock(rwlock_t *rL)
{
    lock(&rL → write_lock);
}
```

```
init_lock(rwlock_t *rL)
{
   init_lock(&rL → read_lock);
   init_lock(&rL → write_lock);
   rL → num_readers = 0;
}

   void write_unlock(rwlock_t *rL)
   {
       unlock(&rL → write_lock);
   }
```

- Write lock behavior is same as the typical lock, only one thread allowed to acquire the lock

Implementation of read-write locks (readers)

```
struct rwlock_t{
    Lock read_lock;
    Lock write_lock;
    int num_readers;
}
void read_lock(rwlock_t *rL)
{
    lock(&rL → read_lock);
    rL → num_readers++;
    if(rL → num_readers == 1)
        lock(&rL → write_lock);
    unlock(&rL → read_lock);
}
```

```
void read_unlock(rwlock_t *rL)
{
    lock(&rL → read_lock);
    rL → num_readers--;
    if(rL → num_readers == 0)
        unlock(&rL → write_lock);
    unlock(&rL → read_lock);
}
```

Implementation of read-write locks (readers)

```
struct rwlock_t{
    Lock read_lock;
    Lock write_lock;
    int num_readers;
}
void read_lock(rwlock_t *rL)
{
   lock(&rL → read_lock);
   rL → num_readers++;
   if(rL → num_readers == 1)
       lock(&rL → write_lock);
   unlock(&rL → read_lock);
}
```

- The first reader acquires the write lock preventing writers to acquire lock
- The last reader releases the write lock to allow writers

```
void read_unlock(rwlock_t *rL)
{
   lock(&rL → read_lock);
   rL → num_readers--;
   if(rL → num_readers == 0)
       unlock(&rL → write_lock);
   unlock(&rL → read_lock);
}
```

CS330: Operating Systems

Condition variables, Concurrency bugs

Condition variables

*pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)*:

Atomically releases the mutex and waits on a condition variable. Resumes execution holding the lock when pthread_cond_signal( ) is invoked.

Important: *The caller should perform the condition check after wakeup*

*pthread_cond_signal (pthread_cond_t *cond)*

Wakes up a waiting thread on condition *cond*, Ideally called holding the mutex.

Example usage

```
cond_t *C; lock_t *L;   bool condition;
```

```
void T1( )
{
    while(1){
      lock(L);
      while(condition != true)
          cond_wait(C, L);
      unlock(L);
      process( );
    }
}
```

```
void T2( )
{
    while(1){
      condition = false;
      process( );
      lock(L);
      condition = true;
      cond_signal(C);
      unlock(L);
    }
}
```

Example usage

```
cond_t *C; lock_t *L;   bool condition;

void T1( )
{
    while(condition != true)
        cond_wait(C, L);
    unlock(L);
    process( );
}
```

```
void T1( )
{
    lock(L);
    condition = true;
    cond_signal(C);
    unlock(L);
}
```

- Why explicit condition check is required (in the waiting thread)?
- Why lock must be held while invoking cond_signal( )?

Example usage

```
cond_t *C; lock_t *L;   bool condition;
```

- Why explicit condition check is required (in the waiting thread)?
- Some implementation on multicore may wake up more than one thread (cause spurious wakeups). For more information, please refer the man page https://linux.die.net/man/3/pthread_cond_wait .
- Why lock must be held while invoking cond_signal( )?

```
    unlock(L);
    process( );
  }
}
```

```
    condition = true;
    cond_signal(C);
    unlock(L);
  }
}
```

Example usage

```
cond_t *C; lock_t *L;   bool condition;
```

- Why explicit condition check is required (in the waiting thread)?
- Some implementation on multicore may wake up more than one thread (cause spurious wakeups). For more information, please refer the man page https://linux.die.net/man/3/pthread_cond_wait .
- Why lock must be held while invoking cond_signal( )?
- The waiting thread may wait indefinitely when the signaling thread executes cond_signal( ) before the waiter invokes cond_wait( )

```
}
}
}
```

Semaphore using condition variables (ostep-31.17)

```
struct sem_t {
   int value;
   lock_t lock;
   cond_t cond;
};
void sem_wait( sem_t *S)
{
    lock(&S → lock);
    while( S → value <= 0)
        cond_wait(&S → cond, &S → lock);
    S → value --;
   unlock(&S → lock);
}
```

```
void sem_init(sem_t *S, int val)
{
    S → value = val;
    cond_init(&S → cond);
    lock_init(&S → lock);
}
void sem_post( sem_t *S)
{
    lock(&S → lock);  S → value ++;
    cond_signal(&S→ cond);
    unlock(&S → lock);
}
```

Concurrency bugs - atomicity issues

```
char *ptr;  // Allocated before use
void T1( )
{
    ...
    strcpy(ptr, “hello world!”);
   …
}
```

```
void T2( )
{
    ...
     if(some_condition)
          free(ptr);
    ...
}
```

- This code is buggy. What is the issue?

Concurrency bugs - atomicity issues

```
char *ptr;  // Allocated before use
void T1( )
{
    ...
    strcpy(ptr, “hello world!”);
   …
}
```

```
void T2( )
{
    ...
     if(some_condition)
          free(ptr);
    ...
}
```

- This code is buggy. What is the issue?
- T2 can free the pointer before T1 uses it.
- How to fix it?

Concurrency bugs - atomicity issues

```
char *ptr;  // Allocated before use
void T1( )
{
    ...
    if(ptr)  strcpy(ptr, “hello world!”);
   …
}
```

```
void T2( )
{
    ...
     if(some_condition)
          free(ptr);
    ...
}
```

- Does the above fix (checking ptr in T1) work?

Concurrency bugs - atomicity issues

```
char *ptr;  // Allocated before use
void T1( )
{
    ...
    if(ptr)  strcpy(ptr, “hello world!”);
   …
}
```

```
void T2( )
{
    ...
     if(some_condition)
          free(ptr);
    ...
}
```

- Does the above fix (checking ptr in T1) work?
- Not really. Consider the following order of execution:
- T1: “if(ptr)” T2: “free(ptr)” T1: “strcpy” Result: Segfault

Concurrency bugs - ordering issues

```
1.  bool pending;
2.  void T1( )
3.  {
4.      pending = true;
5.      do_large_processing( );
6.      while (pending);
7.  }
```

```
1.  void T2( )
2.  {
3.      do_some_processing( );
4.      pending = false;
5.      some_other_processing( );
6.  }
```

- This code works with the assumption that line#4 of T2 is executed after line#4 of T1
- If this ordering is violated, T1 is stuck in the while loop

Concurrency bugs - deadlocks

```
struct acc_t{
        lock_t *L;
        id_t  acc_no;
        long balance;
}
void txn_transfer( acc_t *src,
                   acc_t *dst,  long amount)
{
    lock(src → L);  lock(dst → L);
    check_and_transfer(src, amount);
    unlock(dst → L); unlock(src → L);
}
```

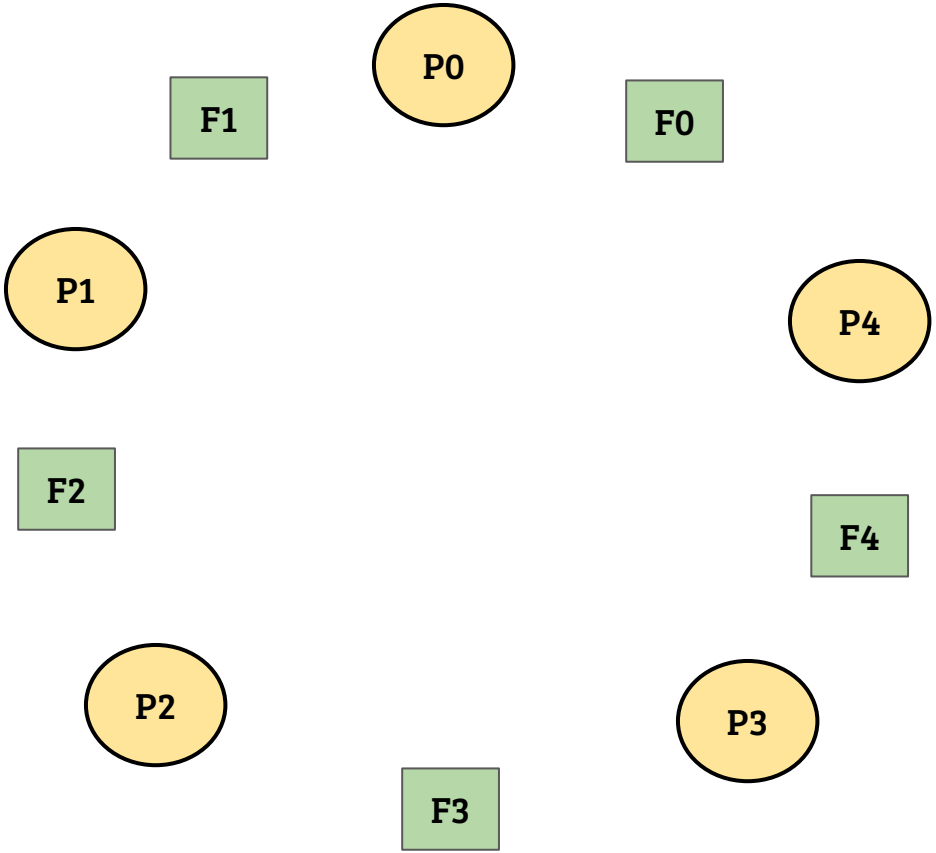
- Consider a simple transfer transaction in a bank
- Where is the deadlock?

Concurrency bugs - deadlocks

```
struct acc_t{
        lock_t *L;
        id_t  acc_no;
        long balance;
}
void txn_transfer( acc_t *src,
                        acc_t *dst,  long amount)
{
    lock(src → L);  lock(dst → L);
    check_and_transfer(src, amount);
    unlock(dst → L); unlock(src → L);
}
```

- Consider a simple transfer transaction in a bank
- Where is the deadlock?
- T1: txn_transfer(iitk, cse, 10000)
  - lock (iitk), lock (cse)
- T2: txn_transfer(cse, iitk, 5000)
  - lock (cse), lock(iitk)

Dining philosophers




```
atomic_t forks[5];
Philosopher( int id)
{
   while (1) {
      think( );
      acquire(forks[id]);
      acquire(forks[(id+1) % 5]);
      eat( );
      release( forks[(id+1) % 5]);
      release(forks[id]);
   }
}
```

Conditions for deadlock

- Mutual exclusion: exclusive control of resources (e.g, thread holding lock)
- Hold-and-wait: hold one resource and wait for other
- No resource preemption: Resources can not be forcibly removed from threads holding them
- Circular wait: A cycle of threads requesting locks held by others. Specifically, a cycle in the directed graph G (V, E) where V is the set of processes and (v1, v2) $\in$ E if v1 is waiting for a lock held by v2

All of the above conditions should be satisfied for a deadlock to occur

Solutions for deadlocks

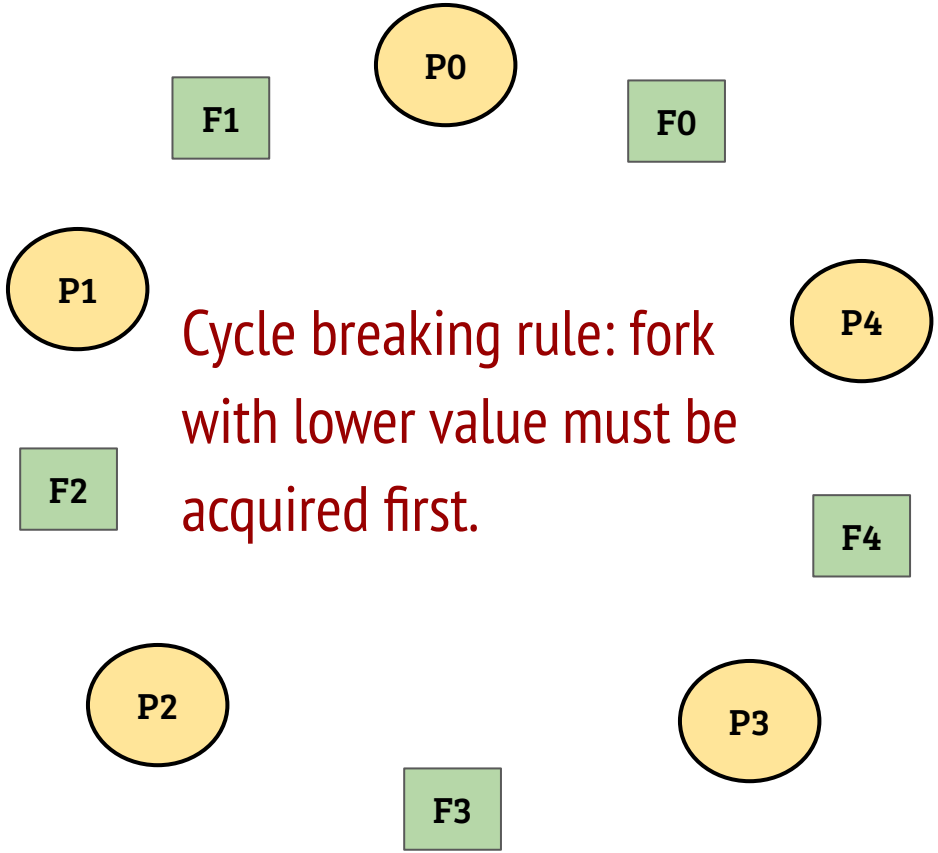
- Remove mutual exclusion: lock free data structures
- Either acquire all resources or no resource
  - trylock(lock) APIs can be used (e.g., pthread_mutex_trylock( ))
- Careful scheduling: Avoid scheduling threads such that no deadlock occur
- Most commonly used technique is to avoid circular wait. This can be achieved by ordering the resources and acquiring them in a particular order from all the threads.

Concurrency bugs - avoiding deadlocks

```
struct acc_t{
        lock_t *L;
        id_t  acc_no;
        long balance;
}
void txn_transfer( acc_t *src,
                   acc_t *dst,  long amount)
{
    lock(src → L);  lock(dst → L);
    check_and_transfer(src, amount);
    unlock(dst → L); unlock(src → L);
}
```

- Deadlock in a simple transfer transaction in a bank
- While acquiring locks, first acquire the lock for the account with lower “acc_no” value
- Account number comparison performed before acquiring the lock

Dining philosophers: breaking the deadlock



Cycle breaking rule: fork with lower value must be acquired first.

```
atomic_t forks[5];
Philosopher( int id)
{
   while (1) {
      if(id == 4){
         acquire(0);
         acquire(4);
      }else{
        acquire(forks[id]);
        acquire(forks[id+1]);
      }
      ...
}
```