Preemption

- The act of forcefully taking the CPU from an application temporarily is called preemption
  - In preemptive scheduling, an application can be preempted
  - Popular OSes implement preemptive scheduling for application threads
- In non-preemptive scheduling, an application is never preempted
  - Some research OSes (e.g., exokernel) use non-preemptive scheduling
  - Why Linux and Windows don't implement non-preemptive scheduling
    - because an application can take the CPU forever

Preemption

- Timer interrupts are essential for preemption

Concurrency

- Heap and global variables are shared among multiple threads

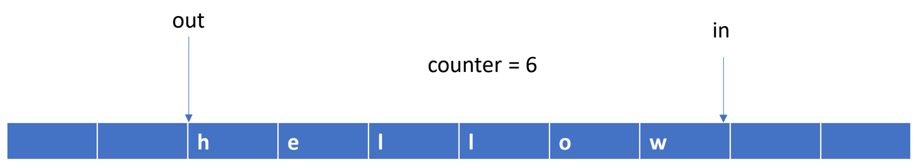
- It is hard to write correct programs when multiple threads can read and write to the same memory
  - We will discuss in the next slides
  - Read section 6.1 from Silberschatz and Galvin
  - Read synchronization from https://faculty.iiitd.ac.in/~piyus/pintos/doc/pintos_6.html#SEC98

Concurrency

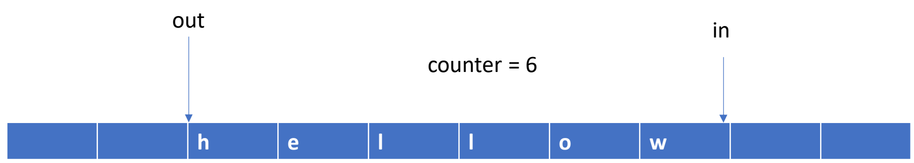
```
int counter = 0;      // global
char buffer[BUFFER_SIZE]; // global
int in = 0, out = 0;

put(char ch) {
   while (counter == BUFFER_SIZE)
         ; // do nothing
    buffer[in] = ch;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}

char get() {
   while (counter ==  0)
         ; // do nothing
    ch = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    return ch;
}
```

In this example, get and put routines are accessing a shared circular queue. The get /put routines remove/insert a character from/into the queue, respectively. The counter contains the total number of elements in the circular queue.

put and get

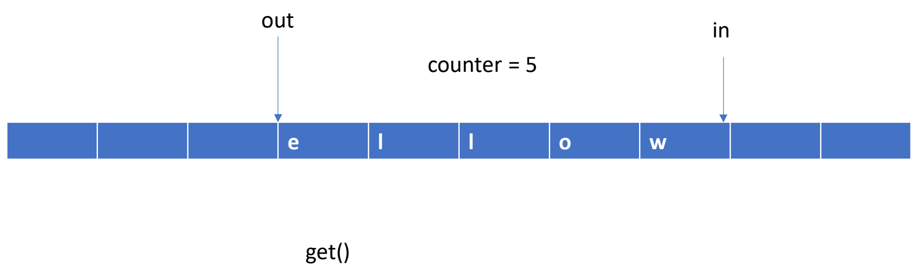


put and get

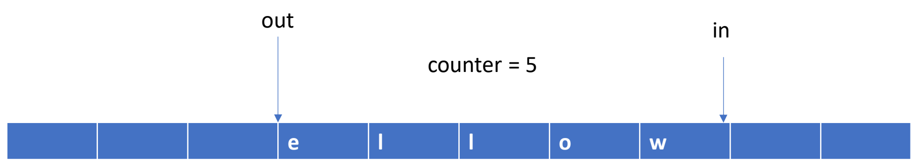


get()

put and get

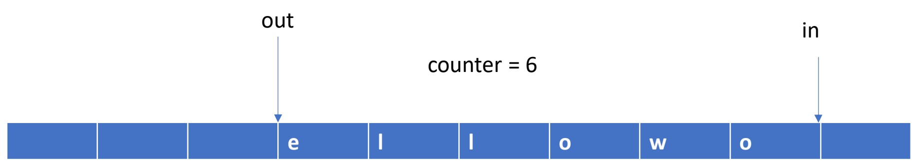


put and get



get()
put('o')

put and get



get()
put('o')

Concurrency

If put and get routines are called at the same time by different threads, then the counter value should remain the same after both the functions finish their execution.

```
int counter = 0;      // global
char buffer[BUFFER_SIZE]; // global
int in = 0, out = 0;

put(char ch) {
   while (counter == BUFFER_SIZE)
          ; // do nothing
    buffer[in] = ch;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}

char get() {
   while (counter ==  0)
          ; // do nothing
    ch = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    return ch;
}
```

Concurrency

Thread1:

counter++;

T1: mov counter, %eax

T1: add $1, %eax

T1: mov %eax, counter

Thread2:

counter--;

T2: mov counter, %eax

T2: sub $1, %eax

T2: mov %eax, counter

Execution:

counter = 6 (initially)

These are the compiler-generated code corresponding to the counter++ and counter-- statements. Now let us see what happens if two threads are active at the same time and trying to update these counters.

Concurrency

Thread1:

counter++;

T1: mov counter, %eax

T1: add $1, %eax

T1: mov %eax, counter

Thread2:

counter--;

T2: mov counter, %eax

T2: sub $1, %eax

T2: mov %eax, counter

Execution:

counter = 6 (initially)

T1: mov counter, %eax

T1: add $1, %eax

T1: mov %eax, counter

T1: schedule

T2: mov counter, %eax

T2: sub $1, %eax

T2: mov %eax, counter

In this case, thread-2 was scheduled when thread-1 had updated the counter. The final value is 6, which is expected. In future slides, whenever there is a switch between T1 to T2 or T2 to T1, we will assume that this is due to schedule.

Concurrency

Thread1:

counter++;

T1: mov counter, %eax
T1: add $1, %eax
T1: mov %eax, counter

Thread2:

counter--;

T2: mov counter, %eax
T2: sub $1, %eax
T2: mov %eax, counter

Execution:

counter = 6 (initially)

T2: mov counter, %eax
T2: sub $1, %eax
T2: mov %eax, counter
T1: mov counter, %eax
T1: add $1, %eax
T1: mov %eax, counter

In this case, thread-1 was scheduled when thread-2 had updated the counter. The final value is 6, which is expected.

Concurrency

Thread1:	Thread2:	Execution:
counter++;	counter--;	counter = 6 (initially)
T1: mov counter, %eax	T2: mov counter, %eax	T2: mov counter, %eax 6
T1: add $1, %eax	T2: sub $1, %eax	T2: sub $1, %eax 5
T1: mov %eax, counter	T2: mov %eax, counter	T1: mov counter, %eax 6
		T1: add $1, %eax 7
		T2: mov %eax, counter 5
		T1: mov %eax, counter 7

However, in this case, thread-1 was scheduled when thread-2 had partially updated the counter. As a consequence, the final value is 7, which is wrong.

Concurrency

Thread1:

counter++;

T1: mov counter, %eax

T1: add $1, %eax

T1: mov %eax, counter

Thread2:

counter--;

T2: mov counter, %eax

T2: sub $1, %eax

T2: mov %eax, counter

Execution:

counter = 6 (initially)

T2: mov counter, %eax 6

T2: sub $1, %eax 5

T1: mov counter, %eax 6

T1: add $1, %eax 7

T1: mov %eax, counter 7

T2: mov %eax, counter 5

In this case, a different sequence of schedules leads to a wrong answer 5.

Race condition

- When multiple threads access the shared memory, then the output of shared memory may depend on the order in which they are accessed
  - When the result indeed depends on the order of execution is called a race condition

Why shared memory?

- What is the point of shared memory?
  - After all, we want isolation among applications

Why shared memory?

- What is the point of shared memory?
  - After all, we want isolation among applications
    - Shared memory enables faster interaction among applications
    - Without shared memory, it won't be easy for applications to interact with each other

How to avoid a race condition

- Identify the sequence of instructions, which may potentially cause a race condition
  - These sequence of instructions are called a critical section
  - A race condition may occur if another thread can execute in the critical section due to a schedule in the critical section

Critical section

Thread1:

counter++;

T1: mov counter, %eax

T1: add $1, %eax

T1: mov %eax, counter

Thread2:

counter--;

T2: mov counter, %eax

T2: sub $1, %eax

T2: mov %eax, counter

Execution:

Here, the logic corresponding updating of counter needs to be in a critical section because a schedule during the partial update of counter can lead to incorrect output.

Locking

- Locking is used to avoid race conditions
- Each critical section is protected using a lock
- A thread acquires the lock on entry to critical section and releases the lock after exiting from the critical section
- Locking ensures that only one thread can execute in the critical sections that are protected using the same lock

Locking

```
struct lock lock_a, lock_b;

acquire(&lock_a);
critical_section-1
release(&lock_a);

acquire(&lock_b);
critical_section-2
release(&lock_b);
```

Is it possible that a schedule in critical_section-1 may cause another thread to execute in critical_section-2?

Yes

Yes, it is possible because both critical sections are protected using different locks.

Locking

```
struct lock lock_a, lock_b;

acquire(&lock_a);
critical_section-1
release(&lock_a);

acquire(&lock_a);
critical_section-2
release(&lock_a);
```

Is it possible that a schedule in critical_section-1 may cause another thread to execute in critical_section-2?

No

No, it is not possible because both critical sections are protected using the same lock.

How to implement a lock?

- Peterson’s solution
  - Section-6.3 from Silberschatz and Galvin

Lock two threads

```
boolean flag[2];
acquire(int i)  {                    // i = current thread's id (0 or 1)
   int j = 1 – i;                    // j = other thread's id
   flag[i] = TRUE;
   while (flag[j]);
}
release(int i) {                     // i = current thread's id
   flag[i] = FALSE;
}
```

This locking solution works for only two threads. The global variable flag contains boolean values corresponding to threads 0 and 1. In the acquire routine, a thread sets its own flag and wait for other thread's flag to become false. In the release routine, a thread resets its own flag.

Lock two threads

```
boolean flag[2];
acquire(int i)  {
   int j = 1 – i;
   flag[i] = TRUE;
   while (flag[j]);
}
release(int i) {
   flag[i] = FALSE;
}
```

T0: flag[0] = TRUE;
T0: acquires lock
T0: schedule
T1: flag[1] = TRUE;
T1: spinning at while
T1: schedule
T0: releases lock, flag[0]=FALSE
T0: schedule
T1: acquires lock

This example shows how acquire and release works.

Lock two threads

```
boolean flag[2];
acquire(int i)  {
   int j = 1 – i;
   flag[i] = TRUE;
   while (flag[j]);
}
release(int i) {
   flag[i] = FALSE;
}
```

T0: flag[0] = TRUE;
T0: schedule
T1: flag[1] = TRUE;
T1: spinning at while
T1: schedule
T0: spinning at while

deadlock

But, this solution is not correct because a particular sequence of schedules can lead to a deadlock. A deadlock is a situation when an application can't make any progress.

Lock two threads

This is the correct implementation, as discussed in class.

```
boolean flag[2];
int turn;
acquire(int i)  {
   int j = 1 – i;
   flag[i] = TRUE;
   turn = j;
   while (flag[j] && turn == j);
}
release(int i) {
   flag[i] = FALSE;
}
```

T0: flag[0] = TRUE;
T0: turn = 1;
T0: schedule
T1: flag[1] = TRUE;
T1: turn = 0;
T1: spinning at while => flag[0] && turn = 0
T1: schedule
T0: acquires lock

In this implementation, a new variable turn is added to eliminate the problem associated with the previous solution. Now, at a given time, turn can have only one value. Due to this, at least one thread will always come out of the while loop. This example shows one particular schedule.

Lock two threads

This is the correct implementation, as discussed in class.

```
boolean flag[2];
int turn;
acquire(int i) {
   int j = 1 – i;
   flag[i] = TRUE;
   turn = j;
   while (flag[j] && turn == j);
}
release(int i) {
   flag[i] = FALSE;
}
```

T0: flag[0] = TRUE;
T0: schedule
T1: flag[1] = TRUE;
T1: turn = 0;
T1: schedule
T0: turn = 1;
T0: schedule
T1: acquires lock
T1: schedule
T0: spinning => flag[1] && turn == 1
T0: schedule
T1: releases lock
T1: flag[1] = FLASE;
T1: schedule
T0: acquires lock

This example shows another possible schedule.

Lock two threads

This is the correct implementation, as discussed in class.

```
boolean flag[2];
int turn;
acquire(int i)  {
   int j = 1 – i;
   flag[i] = TRUE;
   turn = j;
   while (flag[j] && turn == j);
}
release(int i) {
   flag[i] = FALSE;
}
```

If the two threads try to acquire the lock at the same time, then the value of turn can be either 0 or 1. A thread can't set the turn to itself. The thread whose turn is set at the while loop acquires the lock.

schedule

```
struct list_node *ready_list;
struct thread *cur_thread;
void schedule() {
  if (empty(ready_list))
     return;
  list_insert(ready_list, cur_thread);
  struct thread *prev = cur_thread;
  struct thread *next =  get_next_thread(ready_list);
  cur_thread = next;
  context_switch(prev, next);
}
```

Because schedule function is manipulating a global list, if two threads try to update the list simultaneously, the list can go into an inconsistent state.

list_insert(struct thread *t)

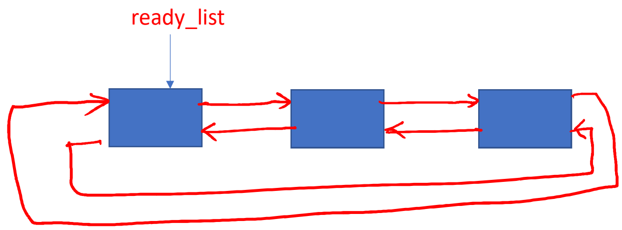
```
node = (struct node*)malloc(sizeof(struct node));
node->t = t;
node->next = ready_list;
node->prev = ready_list->prev;
node->prev->next = node;
node->next->prev = node;
ready_list = node;
```

```
struct node {
   struct thread *t;
   struct node *next;
   struct node *prev;
};
```

This code is assuming that the ready_list has at least one node.

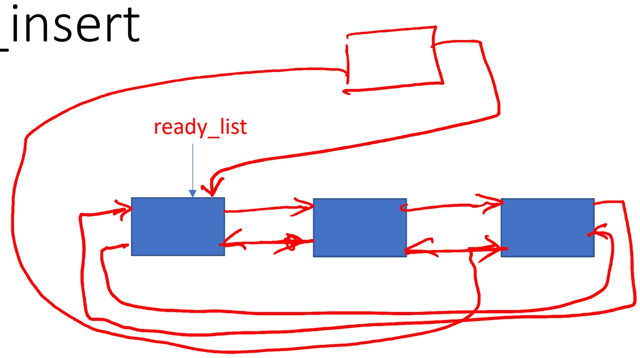
Here is one possible implementation of list_insert.

list_insert



The ready list is a circular doubly linked list. The variable ready_list points to the last element in the ready list.

list_insert



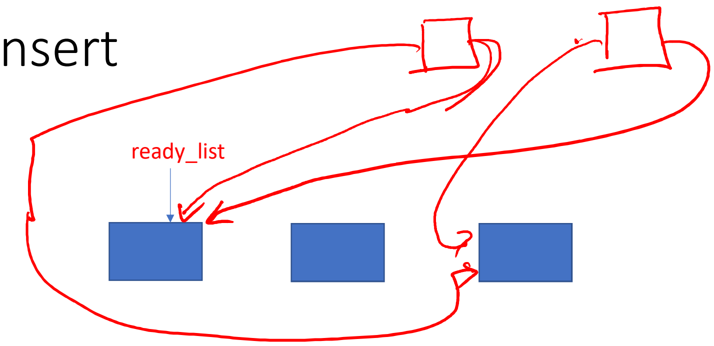
T1: node = malloc()

T1: node->next = ready_list

T1: node->prev = ready_list->prev

If both the threads try to update the linked list, then the final state of list depends on the order in which the schedule was invoked.

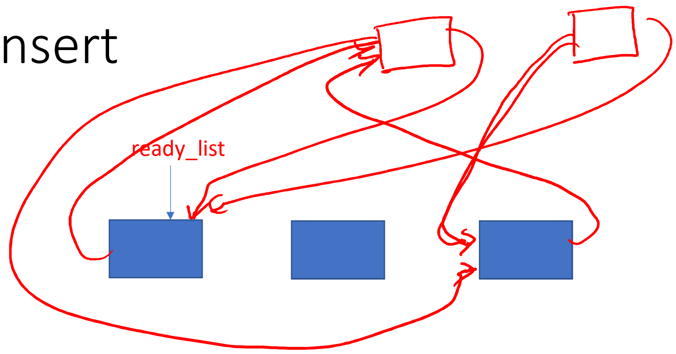
list_insert



T1: node = malloc()
T1: node->next = ready_list
T1: node->prev = ready_list->prev
T2: node = malloc()
T2: node->next = ready_list
T2: node->prev = ready_list->prev

Try yourself to update the list in the order given on this slide.

list_insert

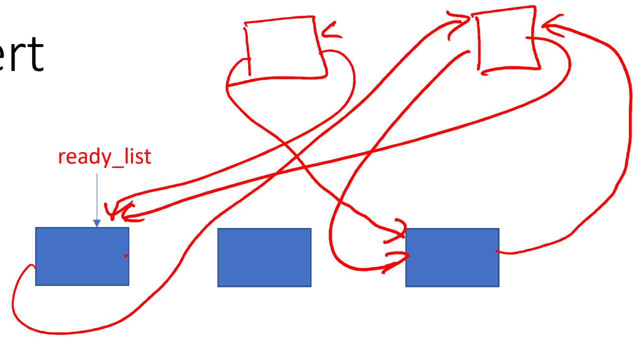


T1: node = malloc()
T1: node->next = ready_list
T1: node->prev = ready_list->prev
T2: node = malloc()
T2: node->next = ready_list
T2: node->prev = ready_list->prev

T1:node->prev->next = node;
T1: node->next->prev = node;
T1: ready_list = node;

Try yourself to update the list in the order given on this slide.

list_insert



T1: node = malloc()
T1: node->next = ready_list
T1: node->prev = ready_list->prev
T2: node = malloc()
T2: node->next = ready_list
T2: node->prev = ready_list->prev

T1:node->prev->next = node;
T1: node->next->prev = node;
T1: ready_list = node;
T2: node->prev->next = node;
T2: node->next->prev = node;
T2: ready_list = node;

Try yourself to update the list in the order given on this slide. In the end, you will encounter that only one element was added to the list.

schedule1

```
schedule1() {
  acquire();
  schedule();
  release();
}
```

To mitigate this problem, instead of calling schedule from the interrupt handler, we call schedule1. schedule1 calls schedule after acquiring a lock and thus prevents multiple threads from updating the ready list.

schedule1

- How to implement the acquire and release
  - Peterson's lock
    - calling a schedule may not be a bad idea while holding the Peterson's lock
      - e.g., when the critical section is large
- Invocation of schedule in the schedule routine itself is a bad idea
  - Wasting of CPU cycles
  - When the critical section is very small, it is okay if we don't let other threads to get scheduled
  - If we use Peterson's lock, then there would be a deadlock

However, schedule is a special routine. We can't use the Peterson's lock, because, on interrupt, the schedule routine will call itself (thus tries to acquire the same lock again and again). This would lead to a deadlock.

Locks for small critical section

```
acquire() {
  status = disable_interrupt();
}

release() {
  set_interrupt_status(status);
}
```

The interrupt flag in the EFLAGS register gives the status of interrupts.

cli instruction can disable the interrupts.

Alternatively, EFLAGS can be changed to enable/disable interrupts.

```
A {
  acquire();
  acquire();
  release();
  release();
```

Another way of implementing lock would be to disable the interrupts. If the interrupts are disabled, then a thread can not be preempted. This lock is useful for small critical sections. For a large critical section, it may introduce a noticeable pause. The acquire routine can also be called from places where the interrupts are already disabled. In the release routine, the interrupts are enabled iff they were enabled during the acquire.

schedule1

```
schedule1() {
  status = disable_interrupt();
  schedule();
  set_interrupt_status(status);
}
```

Here is one possible implementation of schedule1. Notice that now the schedule routine can't be interrupted.

schedule1

```
schedule1() {
  status = disable_interrupt();
  push_list(ready_list, cur_thread);
  schedule();
  set_interrupt_status(status);
}
```

For simplicity, we move the logic corresponding to putting current thread to ready list to schedule1.

schedule

```
struct list_node *ready_list;
struct thread *cur_thread;

void schedule() {
  struct thread *prev = cur_thread;
  struct thread *next =  pop_list(ready_list);
  cur_thread = next;
  context_switch(prev, next);
}
```

This is the modified schedule routine after the previous modification in the schedule1 routine.

Locks

- Disabling interrupts is not suitable for a large critical section
  - An interactive application frequently needs CPU for smooth execution
- There are other ways to implement lock for a large critical section

Is this implementation correct?

```
acquire(struct lock *l) {
  status = interrupt_disable();
  while (l->value == 0);
  l->value = 0;
  set_interrupt_status(status);
}

release(struct lock *l) {
   l->value = 1;
}
```

```
struct lock {
   int value;
};
```

lock is initialized with 1.

Now, let's see how we can implement a lock for large critical section (interrupts are not disabled in the critical section). This implementation may cause a deadlock, as discussed on the next slide.

Is this implementation correct?

```
acquire(struct lock *l) {
  status = interrupt_disable();
  while (l->value == 0);
  l->value = 0;
  set_interrupt_status(status);
}

release(struct lock *l) {
   l->value = 1;
}
```

```
struct lock {
  int value;
};
```

T0: acquire
T0: acquired
T0: schedule
T1: acquire

lock is initialized with 1.

no, because if a thread fails to acquire a lock, it spins while the interrupts are disabled.

Is this implementation correct?

```
acquire(struct lock *l) {
  status = interrupt_disable();
  while (l->value == 0) {
     schedule1();
  }
  l->value = 0;
  set_interrupt_status(status);
}

release(struct lock *l) {
   l->value = 1;
}
```

```
schedule1() {
  status = disable_interrupt();
  push_list(ready_list, cur_thread);
  schedule();
  set_interrupt_status(status);
}
```

lock is initialized with 1.

Yes, this implementation is correct. However, putting the waiting threads to the ready list is not a good idea. Because the scheduler may schedule the waiting threads before the current lock holder releases the lock, this may result in unnecessary wastage of CPU cycles.

Is this implementation correct?

```
acquire(struct lock *l) {
  status = interrupt_disable();
  while (l->value == 0) {
     schedule1();
  }
  l->value = 0;
  set_interrupt_status(status);
}

release(struct lock *l) {
   l->value = 1;
}
```

```
schedule1() {
  status = disable_interrupt();
  push_list(ready_list, cur_thread);
  schedule();
  set_interrupt_status(status);
}
```

lock is initialized with 1.

Yes, but not good because there is no point in scheduling waiting threads until the lock holder releases the lock

acquire

```
acquire(struct lock *l) {
  status = interrupt_disable();
  while (l->value == 0) {
     list_push(l->wait_list, cur_thread);
     schedule();
  }
  l->value = 0;
  set_interrupt_status(status);
}
```

```
struct lock {
  int value;
  struct list *wait_list;
};
```

Instead of putting the threads in the ready list, the lock implementation puts them in another list, thus preventing them from getting scheduled.

A better solution is to put the waiting threads in a different list (other than the ready list). The lock variable also contains a waiting list that contains all the threads which are waiting for acquiring the lock. The threads are moved to the ready list during release. We will discuss this in the next class.