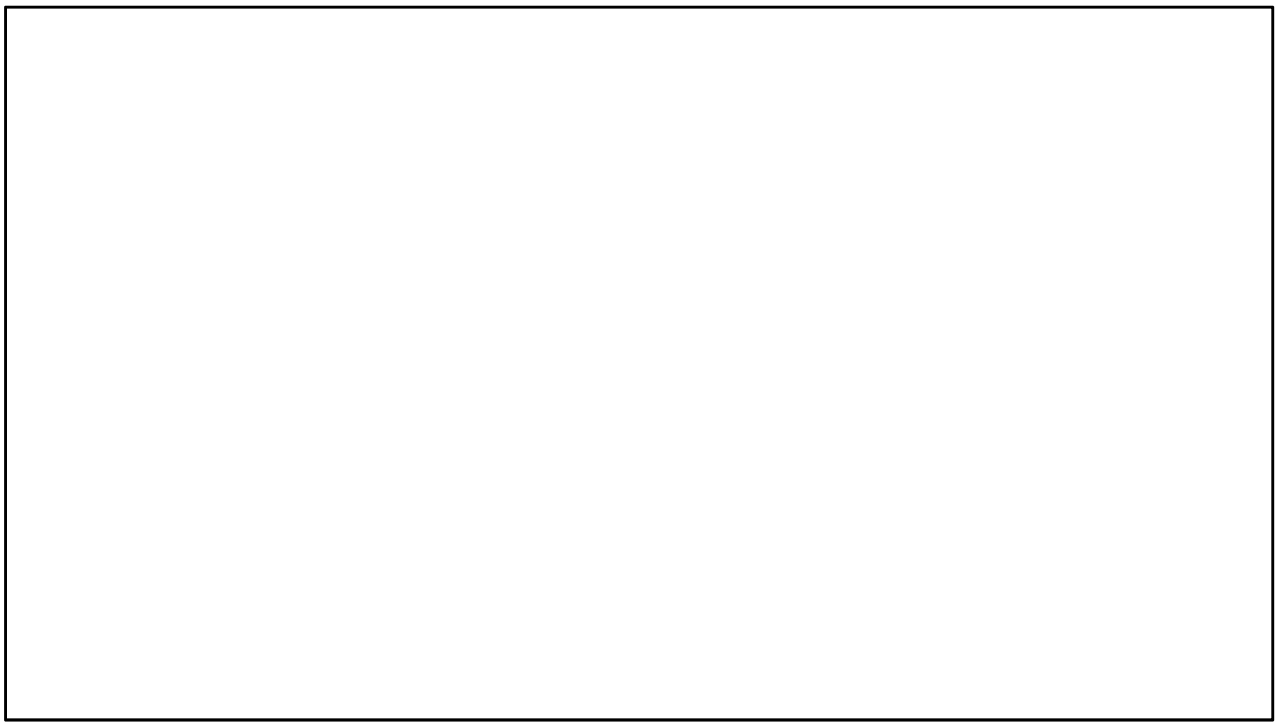


Is this implementation correct?

```
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;
}
```

T1

T2

T3

T2

```
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

The schedule1 routine puts the waiting threads to the ready list such that it can be scheduled later (say on timer interrupt). However, it doesn't make sense to let these waiting threads getting scheduled until the lock holder releases the lock. A better solution would be to put these threads in some other list (not ready list). During release, the lock holder can remove a thread from the waiting list and add them to the ready list.

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.

The lock structure contains a list that contains all the threads that are waiting for acquiring the corresponding lock. In acquire routine, if a thread is unable to acquire a lock, it puts itself to the waiting list of the corresponding lock and call schedule. Notice, schedule is different from schedule1. schedule doesn't put the current thread to the ready list. Also, schedule assumes that the caller must have disabled the interrupt before calling schedule. If interrupts are enabled, then schedule may call itself (unless we implement some other logic to prevent that) and thus end up putting the ready list in an inconsistent state.

acquire

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

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

Is it okay to call schedule with interrupts disabled? Does this mean that the new thread will be running with interrupts disabled?

Even if we are disabling interrupts while calling schedule, it will be enabled after the schedule returns.

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);
}
```

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

Is it okay to call schedule with interrupts disabled? Does this mean that the new thread will be running with interrupts disabled?

No, if the new thread was preempted using schedule1 routine, then the interrupts will be enabled in the schedule1 method, after returning from the schedule.

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);
}
```

Is it okay to call schedule with interrupts disabled? Does this mean that the new thread will be running with interrupts disabled?

If the schedule (for the new thread) was called from the acquire routine, then the interrupt will be enabled at the end of the acquire routine.

release

```
release(struct lock *l) {
  t = list_pop(l->wait_list);
  if (t)
    list_push(ready_list, t);
  l->value = 1;
}
```

```
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);
}
```

What is the problem with the release implementation?

release

```
release(struct lock *l) {
  t = list_pop(l->wait_list);
  if (t)
     list_push(ready_list, t);
  l->value = 1;
}
```

```
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);
}
```

What is the problem with the release implementation?

The release is modifying the wait_list without disabling interrupts. An interrupt may schedule the current thread after the partial update of wait_list, and new thread may call the acquire routine. In the acquire routine, list_push may cause wait_list to go into an inconsistent state.

release

```
release(struct lock *l) {
  t = list_pop(l->wait_list);
  if (t)
     list_push(ready_list, t);
  l->value = 1;
}
```

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

What is the problem with the release implementation?

The release is modifying the ready_list without disabling interrupts. An interrupt may call schedule1 after the partial update of the ready_list. In schedule1 routine, the list_push may cause the ready_list to go into an inconsistent state.

release

```
release(struct lock *l) {
  t = list_pop(l->wait_list);
  if (t)
     list_push(ready_list, t);
  l->value = 1;
}
```

```
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);
}
```

What is the problem with the release implementation?

Let us say that list_pop returns NULL. At this point, an interrupt may schedule a new thread (T2) that may call the acquire routine and add itself to wait_list. Thread T2 will never be woken up unless a new thread tries to acquire the same lock.

This case is described on the next slide.

release

```
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);
}

release(struct lock *l) {
  t = list_pop(l->wait_list);
  if (t)
     list_push(ready_list, t);
  l->value = 1;
}
```

T1: try to acquire lock A
T1: acquired lock A
T1: try to release lock A
T1: list_pop returns NULL
T1: schedule1 is called
T2: try to acquire lock A
T2: added to wait_list, schedule1 was called
T1: released lock A
…
T2: is still in the wait list ..
…
T3: try to acquire lock A, acquired lock A, try to release lock A, add T2 to ready list

release

```
release(struct lock *l) {
  status = interrupt_disable();
  t = list_pop(l->wait_list);
  if (t)
     list_push(ready_list, t);
  l->value = 1;
  set_interrupt_status(status);
}
```

The correct implementation of release disables the interrupts throughout the function body.

release

```
release(struct lock *l) {
  status = interrupt_disable();
  t = list_pop(l->wait_list);
  if (t)
     list_push(ready_list, t);
  l->value = 1;
  set_interrupt_status(status);
}
```

```
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);
}
```

What is the scenario in which the thread that was added to the ready list (i.e., t) may not get the lock after getting schedule?

release

```
release(struct lock *l) {
  status = interrupt_disable();
  t = list_pop(l->wait_list);
  if (t)
     list_push(ready_list, t);
  l->value = 1;
  set_interrupt_status(status);
}
```

```
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);
}
```

What is the scenario in which the thread that was added to the ready list (i.e., t) may not get the lock after getting schedule?

If another thread acquires the lock between the release of lock and scheduling of t.

Let us say thread T1 was releasing the lock. In release, T1 found T2 on the waiting list, added them to the ready list, and then released the lock (returned from the release routine). Now, before T2 gets CPU, another thread T3 was scheduled and acquired the lock. After this, T3 was preempted (before releasing the lock), and T2 was scheduled. In this case, T2 will not get the lock, and it waits again by putting itself to the waiting list and calling schedule.

Locks

- Locks are an example of synchronization primitive
- Another example of synchronization primitive is a semaphore

Semaphores

- A semaphore consists of an integer (value) with two operators down and up that manipulates the value in a critical section
- down
  - wait until the value becomes positive
  - decrement the value
- up
  - increment the value

Semaphores are the same as locks, just that the value of semaphore can be more than one. Semaphores can be used to implement locks and may be used in other scenarios as well.

down

```
down(struct semaphore *s) {
  status = interrupt_disable();
  while (s->value <= 0) {
     list_push(s->wait_list, cur_thread);
     schedule();
  }
  s->value--;
  set_interrupt_status(status);
}
```

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

Same as acquire, except the value can be more than 1.

up

```
up(struct semaphore *s) {
  status = interrupt_disable();
  t = list_pop(s->wait_list);
  if (t)
     list_push(ready_list, t);
  s->value++;
  set_interrupt_status(status);
}
```

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

Same as release, except the value can be more than 1.

Locking using semaphore

```
lock_init(semaphore *s) {
  s->value = 1;
}

lock_acquire(semaphore *s) {
  down(s);
}
lock_release(semaphore *s) {
  up(s);
}
```

We can implement locks by initializing the semaphore to 1.

Semaphores

- Parent wait for child to get scheduled

```
parent() {
   struct semaphore s;
   s.value = 0;
   create_thread(child, &s);
   down(&s);
}
```

```
child(struct semaphore *s) {
   up(s);
}
```

Semaphores can also be used to implement wait and signal. If a thread initializes the semaphore with zero and calls down, then it will wait until somebody signals by calling up. In this example, a parent thread creates a child thread and passes a semaphore to the child. The semaphore is initialized with zero. The parent is waiting for the child to send a signal when it gets a chance to run.

Condition variables

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

```
void put(char ch) {
   while (counter == BUFFER_SIZE);
    buffer[in] = ch;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
char get() {
   while (counter ==  0);
    char ch = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    return ch;
}
```

This is the same example that we have discussed before. buffer is a circular queue. If the queue is not full, the put routine inserts a character to the buffer. If the buffer is not empty, the get routine removes a character from the queue. Multiple threads can be in the get and put routines at the same time.

Condition variables

```
/* GLOBALS */
int counter = 0;
char buffer[BUFFER_SIZE];
int in = 0, out = 0;
struct condition not_full, not_empty;
```

```
void put(char ch) {
   while (counter == BUFFER_SIZE);
    buffer[in] = ch;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
char get() {
   while (counter ==  0);
    char ch = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    return ch;
}
```

One of the problems with this code is waiting for the buffer to become not empty or not full in a busy loop. Notice that rather than waiting in the while loop in the put routine, it is better to call yield because the buffer can only become not full if the get routine gets a chance to run. The put routine should not be scheduled until a thread executing get routine decrements the counter. In this case, the condition variables can be used instead of waiting for a condition to be true in a busy loop.

Condition variables

- cond_wait(struct condition *cond)
  - sleep if the condition is not true
- cond_signal(struct condition *cond)
  - wakeup a thread that is sleeping on this condition

cond_wait and cond_signal take a condition variable as input. cond_wait is called when a condition is not true. It puts the current thread to sleep. cond_signal wakes up a thread that is sleeping on the input condition variable. After a thread (waiting on condition variable) wakes up, it rechecks the condition. If the condition is not true, it goes to sleep mode again.

Condition variables

```
/* GLOBALS */
int counter = 0;
char buffer[BUFFER_SIZE];
int in = 0, out = 0;
struct condition not_full, not_empty;
```

```
void put(char ch) {
   while (counter == BUFFER_SIZE)
      cond_wait(&not_full);
    buffer[in] = ch;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
    cond_signal(&not_empty);
}
```

```
char get() {
   while (counter ==  0)
      cond_wait(&not_empty);
    char ch = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    cond_signal(&not_full);
    return ch;
}
```

This example can be rewritten using cond_wait and cond_signal like this.

Condition variables

```
/* GLOBALS */
int counter = 0;
char buffer[BUFFER_SIZE];
int in = 0, out = 0;
struct condition not_full, not_empty;
```

```
void put(char ch) {
  acquire(&lock);
   while (counter == BUFFER_SIZE)
      cond_wait(&not_full);
   buffer[in] = ch;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
   cond_signal(&not_empty);
   release(&lock);
}
```

```
char get() {
  acquire(&lock);
   while (counter ==  0)
      cond_wait(&not_empty);
   char ch = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   counter--;
   cond_signal(&not_full);
   release(&lock)
   return ch;
}
```

There is another problem with this code. We need locks before accessing the counters, as we have discussed earlier. If we add locks at the start and end of put and get routines, there could be a deadlock. For example, a thread that is calling the put routine can go to sleep if the buffer is full while holding the lock.

Condition variables

```
/* GLOBALS */
int counter = 0;
char buffer[BUFFER_SIZE];
int in = 0, out = 0;
struct condition not_full, not_empty;
```

```
void put(char ch) {
    acquire(&lock);
    while (counter == BUFFER_SIZE) {
        release(&lock);
        cond_wait(&not_full);
        acquire(&lock);
    }
    buffer[in] = ch;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
    cond_signal(&not_empty);
    release(&lock);
}
```

```
char get() {
    acquire(&lock);
    while (counter == 0) {
        release(&lock);
        cond_wait(&not_empty);
        acquire(&lock);
    }
    char ch = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    cond_signal(&not_full);
    release(&lock)
    return ch;
}
```

To eliminate the deadlocks, we can release the lock before sleeping and acquire it back after waking up. But this could lead to the lost wakeup problem. For instance, if a thread (say T1) got preempted just before the cond_wait in get, and some other thread (say T2) starts executing the put routine and inserts a character in the buffer. Now the cond_signal in T2 is not going to wake up any thread (because T1 hasn't called the cond_wait yet). If the scheduler schedules T1 after some time, it would put itself to the sleeping list even though the buffer is not empty. This is also called the lost wakeup problem. One of the reasons why T2 got scheduled before T1 was added to the sleeping list because T2 has released the lock before calling cond_wait. To solve this problem, cond_wait takes a lock as an argument and expects the caller to acquire the lock before the call. cond_wait releases the lock after adding the current thread to the sleeping list. cond_signal expects the caller to acquire the same lock (as in cond_wait) before the call. cond_signal moves a thread from the sleeping list to the ready list.

Condition variables

- cond_wait(struct condition *cond, struct lock *lock)
  - sleep if the condition is not true
  - the lock must be acquired before calling this routine
  - releases the lock after adding the thread to the sleeping list
  - reacquires the lock before returning
- cond_signal(struct condition *cond, struct lock *lock)
  - wakeup a thread that is sleeping on this condition
  - the lock must be acquired before calling this routine

Condition variables

```
/* GLOBALS */
int counter = 0;
char buffer[BUFFER_SIZE];
int in = 0, out = 0;
struct condition not_full, not_empty;
```

```
void put(char ch) {
   acquire(&lock);
   while (counter == BUFFER_SIZE) {
      cond_wait(&not_full, &lock);
    }
    buffer[in] = ch;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
    cond_signal(&not_empty, &lock);
    release(&lock);
}
```

```
char get() {
   acquire(&lock);
   while (counter ==  0) {
      cond_wait(&not_empty, &lock);
    }
    char ch = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    cond_signal(&not_full, &lock);
    release(&lock)
    return ch;
}
```

The final correct code looks like this.

Condition variables

- cond_wait
  - called while the lock is held
  - releases the lock after adding itself to wait_list
  - yields CPU
  - reacquires the lock after getting rescheduled
- cond_signal
  - called while the lock is held
  - move a thread from the wait_list to ready_list

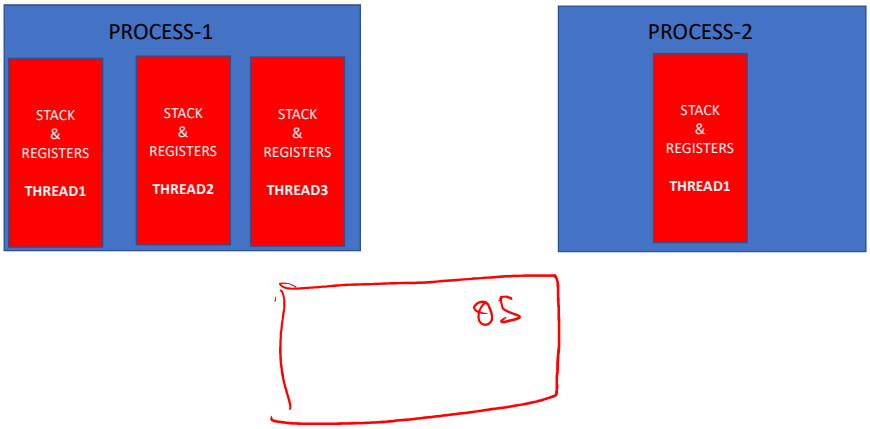
Isolation

- A thread can access data of other threads
- We don’t want an application to see data of other applications
  - Otherwise, a malicious application can steal our sensitive data
    - e.g., password from a banking application
- We need a different abstraction for isolating applications

Process

- A process is a container that contains one or more threads
- Processes can’t see data of each other
- OS is shared between all the processes
  - As of now, we know that the interrupt handler is the part of OS
  - Processes can’t see the data of OS

Process

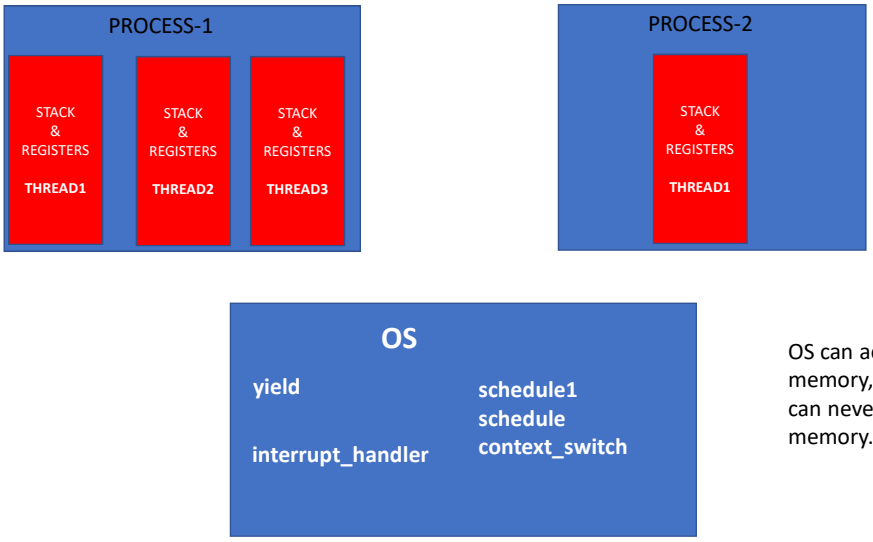

PROCESS-1
STACK & REGISTERS THREAD1
STACK & REGISTERS THREAD2
STACK & REGISTERS THREAD3
PROCESS-2
STACK & REGISTERS THREAD1
OS

Applications

- An application is a group of processes
  - E.g., each tab in the Firefox application could be a process
- Applications can have multiple threads
  - E.g., each tab in the Firefox application could be a thread
- Developers can choose between multiple threads, multiple processes, and a combination of both
  - multiple threads enable faster communication
  - multiple processes enable stronger isolation

If we have to choose between multiple processes vs. multiple threads for developing an application, no choice is better than another in all aspects. Threads may be better for performance (because of faster communication using shared memory), but not for bug isolation. If a thread crashes, everything else (other threads) crashes. This may not be true for processes. But processes may end up using more system resources than threads.

OS



OS can access a process's memory, but the process can never access the OS memory.

Memory isolation

- Will discuss later

OS

- So far we have discussed that an application thread can yield a CPU
  - by calling a yield routine in OS
- Interrupt handler is automatically invoked by the hardware after a timer interrupt
- Can we allow an application to call yield but disallow other routines?
  - What can go wrong if the application can call schedule directly?

OS

- So far we have discussed that an application thread can yield a CPU
  - by calling a yield routine in OS
- Interrupt handler is automatically invoked by the hardware after a timer interrupt
- Can we allow an application to call yield but disallow other routines?
  - What can go wrong if the application can call schedule directly?
    - If it happens, the schedule function will run with interrupts enabled and could potentially make the ready list inconsistent

How can we restrict applications from calling schedule directly?

- No way, using jmp instruction application can jump to anywhere

- What if we disallow assembly code?
  - We can compile OS as a library that only exports the yield routine
  - The loader (OS) can patch the call to yield in the application binary with the real address of yield at load time

Function pointers in C

```
typedef int (*fooptr)(int, int);

int foo(int x, int y) {
  return x + y;
}

int main() {
   fooptr ptr = foo;
   return ptr(10, 11);
}
```

A function pointer is a variable that stores the address of a function. A function can be indirectly called using the address stored in a function pointer. Along with other usage, function pointers are used in the generic implementation of library routines. For example, a generic implementation of the quicksort algorithm can take a user-defined function as an argument to compare two inputs.

Function pointers in C

```
typedef int (*fooptr)(int, int);

int foo(int x, int y) {
  return x + y;
}

int main() {
   fooptr ptr = (fooptr)0x10123;
   return ptr(10, 11);
}
```

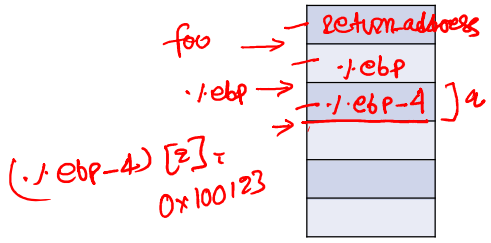
In the C language, any address can be stored in a function pointer. C applications can jump to any address by storing it to a function pointer and calling it indirectly.

How can we restrict applications from calling schedule directly?

- What if we disallow the function pointers?

Applications can modify the return address on the stack

```
foo() {
  unsigned *a = (unsigned*)&a;
  a[2] = 0x100123;
}
```



The application can overwrite the return address on the stack to jump to any address. In this example, when foo returns, the CPU will jump to address 0x100123.

How about Java?

- No pointers
- No out of bounds array accesses
- Automatic memory management
- No arbitrary typecasts
- Yes, for Java applications we can enforce this restriction

How can we restrict applications from calling schedule directly?

- The goal of the OS to allow all kind of languages including assembly
- We need some sort of hardware support for this
- Let us assume for now that if an application directly jumps to OS routines, the OS kills the application