CSE 306 Operating Systems
Kernel Synchronization

YoungMin Kwon

SUNY Korea
The State University of New York

Kernel Synchronization Methods

- Atomic operations
- Spin locks
- Reader-writer spin locks
- Semaphores
- Reader-writer semaphores

SUNY Korea
The State University of New York

Atomic operations

- Atomic operations provide instructions that execute atomically
  - Without interruptions

- A simple race condition (i = i + 1)

Thread 1	Thread 2
get i (7)	get i (7)
increment i (7 -> 8)	
—	increment i (7 -> 8)
write back i (8)	—
—	write back i (8)

SUNY Korea

Atomic operations

- Atomic operators can fix the race condition

Thread 1	Thread 2
get, increment, and store i (7 -> 8)	—
—	get, increment, and store i (8 -> 9)

Thread 1	Thread 2
—	get, increment, and store i (7 -> 8)
get, increment, and store i (8 -> 9)	—

Atomic operations

```
typedef struct {
    volatile int counter;
} atomic_t;


atomic_t v;                        // define v
atomic_t u = ATOMIC_INIT(0);       // define u and initialize it to 0


atomic_set(&v, 4);                 // v = 4     (atomically)
atomic_add(2, &v);                 // v = v + 2 (atomically)
atomic_inc(&v);                    // v = v + 1 (atomically)


printk("%d\n", atomic_read(&v));// print v
```

SUNY Korea

```c
//  At declaration, initialize to i.
ATOMIC_INIT(int i)

//  Atomically read the integer value of v.
int atomic_read(atomic_t *v)
//  Atomically set v equal to i.
void atomic_set(atomic_t *v, int i)

//  Atomically add i to v.
void atomic_add(int i, atomic_t *v)
//  Atomically subtract i from v.
void atomic_sub(int i, atomic_t *v)
//  Atomically add one to v.
void atomic_inc(atomic_t *v)
//  Atomically subtract one from v.
void atomic_dec(atomic_t *v)

//  Atomically subtract i from v and return true if
//  the result is zero; otherwise false.
int atomic_sub_and_test(int i, atomic_t *v)
//  Atomically add i to v and return true if
//  the result is negative; otherwise false.
int atomic_add_negative(int i, atomic_t *v)
```

SUNY Korea

```
//  Atomically add i to v and return the result.
int atomic_add_return(int i, atomic_t *v)
//  Atomically subtract i from v and return the result.
int atomic_sub_return(int i, atomic_t *v)
//  Atomically increment v by one and return the result.
int atomic_inc_return(int i, atomic_t *v)
//  Atomically decrement v by one and return the result.
int atomic_dec_return(int i, atomic_t *v)

//  Atomically decrement v by one and return true if zero;
//  false otherwise.
int atomic_dec_and_test(atomic_t *v)
//  Atomically increment v by one and return true if
//  the result is zero; false otherwise.
int atomic_inc_and_test(atomic_t *v)
```

SUNY Korea

Atomic Bitwise Operations

```
unsigned long word = 0;

set_bit(0, &word);      // bit zero is now set (atomically)
set_bit(1, &word);      // bit one is now set  (atomically)
printk("%ul\n", word); // will print "3"

clear_bit(1, &word);    // bit one is now unset (atomically)
change_bit(0, &word);   // bit zero is flipped; now it is unset (atomically)

// atomically sets bit zero and returns the previous value (zero)
if (test_and_set_bit(0, &word)) {
    // never true ...
}

// the following is legal; you can mix
// atomic bit instructions with normal C
word = 7;
```

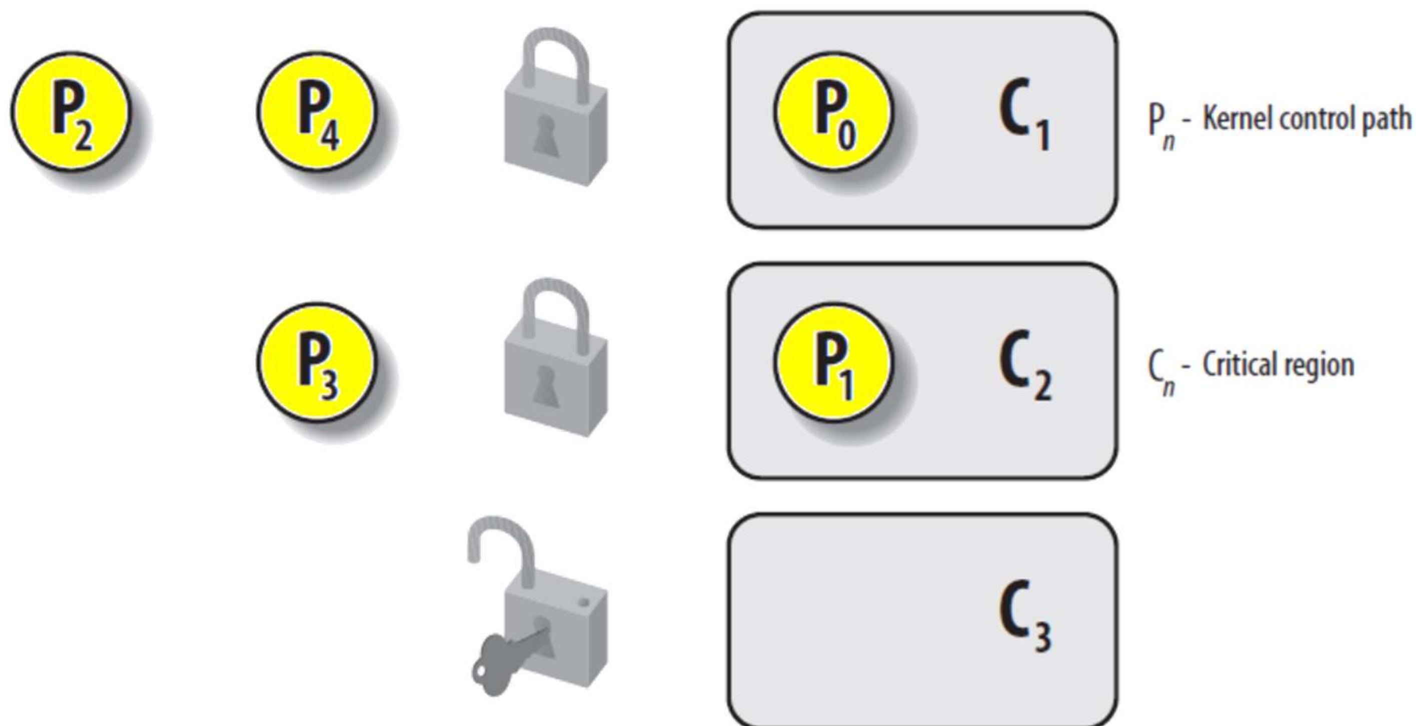
SUNY Korea

```c
// Atomically set the nr-th bit starting from addr.
void set_bit(int nr, void *addr)
// Atomically clear the nr-th bit starting from addr.
void clear_bit(int nr, void *addr)
// Atomically flip the value of the nr-th bit starting from addr.
void change_bit(int nr, void *addr)

// Atomically set the nr-th bit starting from addr
// and return the previous value.
int test_and_set_bit(int nr, void *addr)
// Atomically clear the nr-th bit starting from addr
// and return the previous value.
int test_and_clear_bit(int nr, void *addr)
// Atomically flip the nr-th bit starting from addr
// and return the previous value.
int test_and_change_bit(int nr, void *addr)

// Atomically return the value of the nr-th bit starting from addr.
int test_bit(int nr, void *addr)
```

SUNY Korea

Spin Locks

- Protecting critical regions with several locks



SUNY Korea

Spin Locks

- Properties
  - A spin lock can be held (contended) by at most one thread of execution
  - If a thread tries to acquire an already contended lock, the thread busy loops (spins)
    - Locks should be held for a short duration
  - If the lock is not contended, the thread can immediately acquire the lock and continue

Spin Locks

- Properties
  - Spin locks are not recursive
    - If a thread attempts to acquire a lock that it already owns, the thread will spin.
  - If a spin lock is held, the kernel becomes NON-preemptive
- Kernel Preemption
  - In general, the kernel is preemptive
    - The kernel can stop running at any instant to enable a process of higher priority to run
  - Planned process switch: voluntary sleep
  - Forced process switch: interrupt → higher priority process

Spin Locks

- Basic usage

```
DEFINE_SPINLOCK(mr_lock)

spin_lock(&mr_lock);
// critical region...
spin_unlock(&mr_lock);
```

- On uniprocessor systems
  - The locks compile away: they will disable/enable kernel preemption

SUNY Korea

Spin Locks in Interrupt Handlers

- Spin locks can be used, but not semaphores
  - Interrupt context
- Using a lock in an interrupt handler and in a non-interrupt handler.
  - Disable local interrupts before acquiring the lock
  - Otherwise ⇒ deadlock
    - Your interrupt handler will spin to acquire the lock
    - The lock holder does not run until the interrupt handler completes

SUNY Korea
The State University of New York

Spin Locks in Interrupt Handlers

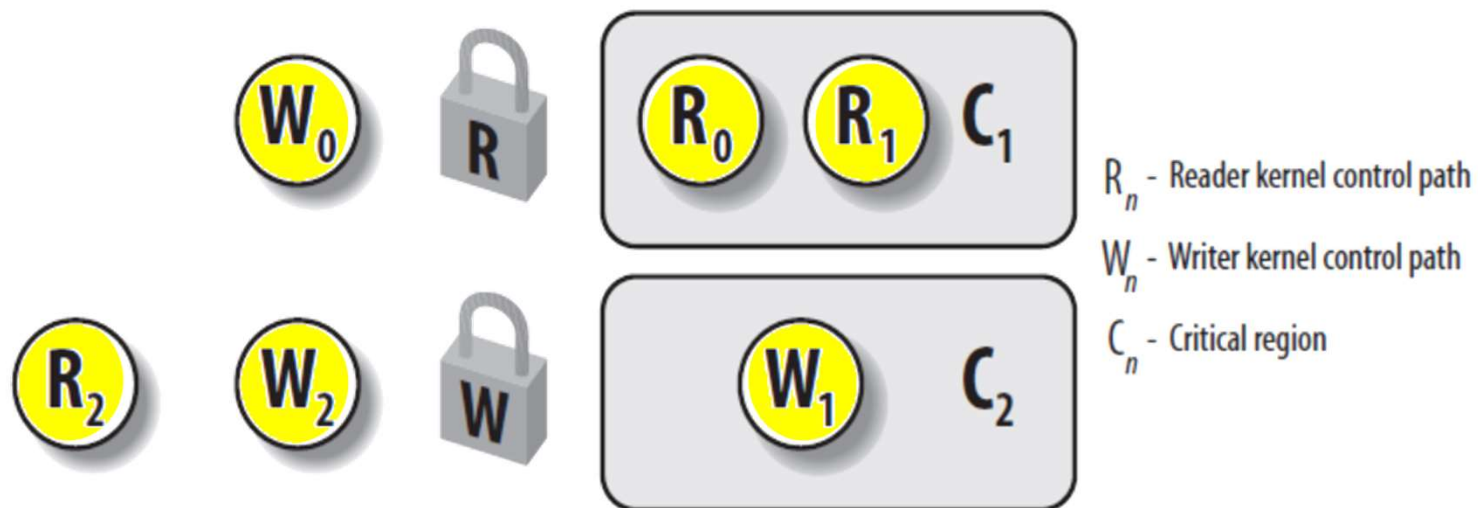
```
DEFINE_SPINLOCK(mr_lock)

// interrupts will be disabled
spin_lock_irq(&mr_lock);

// critical region...

// interrupts will be enabled
spin_unlock_irq(&mr_lock);
```

```
DEFINE_SPINLOCK(mr_lock)
unsigned long flags;

// current interrupt enabled status
// will be saved at flags;
// interrupts will be disabled
spin_lock_irqsave(&mr_lock, flags);

// critical region...

// the previous interrupt enabled
// status will be restored
spin_unlock_irqsave(&mr_lock, flags);
```

SUNY Korea
The State University of New York

Reader-Writer Spin Locks

- Properties
  - One or more readers can hold the reader lock
  - At most one writer can hold the writer lock with no concurrent readers



SUNY Korea

Reader-Writer Spin Locks

- Properties
  - The same thread can recursively acquire the reader lock
  - If your interrupt routine has only readers but no writers
    - You can use read_lock() instead of read_lock_irqsave()
    - But, you need to use write_lock_irqsave() to disable local interrupts

Reader-Writer Spin Locks

- Basic usage

```
DEFINE_RWLOCK(mr_rwlock);

// in the reader code path
read_lock(&mr_rwlock);
// critical section (read only) ...
read_unlock(&mr_rwlock);

// in the writer code path
write_lock(&mr_rwlock);
// critical section (read and write) ...
write_unlock(&mr_lock);
```

SUNY Korea
The State University of New York

Semaphores

- Semaphores are sleeping locks
  - If a task attempts to acquire an unavailable semaphore, it is put in a wait queue
  - When the semaphore becomes available, one of the tasks in the wait queue is awaken

Semaphores

- Using semaphores
  - To hold the lock for a long time
    - Overhead (sleeping, maintaining the wait queue, waking up) is too large for holding a lock for only a short duration
  - Semaphores should be used only in the process context, but NOT in the interrupt context
    - Interrupt context is not schedulable

SUNY Korea
The State University of New York

Semaphores

- Using semaphores
  - You can sleep while holding a semaphore
    - Not a deadlock situation: other process, tries to acquire the semaphore will go to sleep
  - You SHOULD NOT hold a spin lock when you are acquiring a semaphore
    - You should not sleep while holding a spin lock
    - Because process preemption is disabled while holding a spin lock, and other process will spin forever

SUNY Korea
The State University of New York

Semaphores

- Basic usage

```
// define and initialize a semaphore, named mr_sem, with a count of one
static DEFINE_SEMAPHORE(mr_sem);
// struct semaphore mr_sem;       // you can also define and initialize a
// sema_init(&mr_sem, count);     // semaphore this way

// attempt to acquire the semaphore ...
if (down_interruptible(&mr_sem)) {
   // -EINTER on receiving a signal,
   // semaphore is not acquired ...
}

// critical region ...

// release the given semaphore
up(&mr_sem);
```

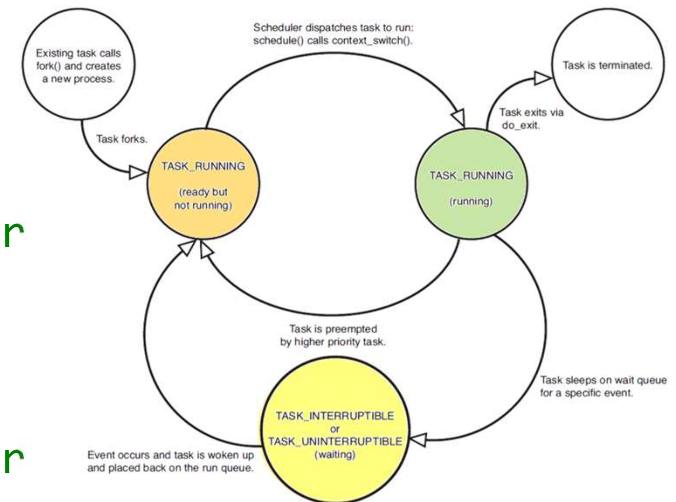
SUNY Korea

Semaphore Methods

```
// Initializes the dynamically created semaphore to the given count
sema_init(struct semaphore *, int)

// Initializes the dynamically created semaphore with a count of one
init_MUTEX(struct semaphore *)

// Initializes the dynamically created semaphore with a count of zero
// (so it is initially locked)
init_MUTEX_LOCKED(struct semaphore *)
```

SUNY Korea
The State University of New York

Semaphore Methods




```
// Tries to acquire the given semaphore and enter
// interruptible sleep if it is contended
down_interruptible(struct semaphore *)

// Tries to acquire the given semaphore and enter
// uninterruptible sleep if it is contended
down(struct semaphore *)

// Tries to acquire the given semaphore and immediately return nonzero
// if it is contended
down_trylock(struct semaphore *)

// Releases the given semaphore and wakes a waiting task, if any
up(struct semaphore *)
```

SUNY Korea
The State University of New York

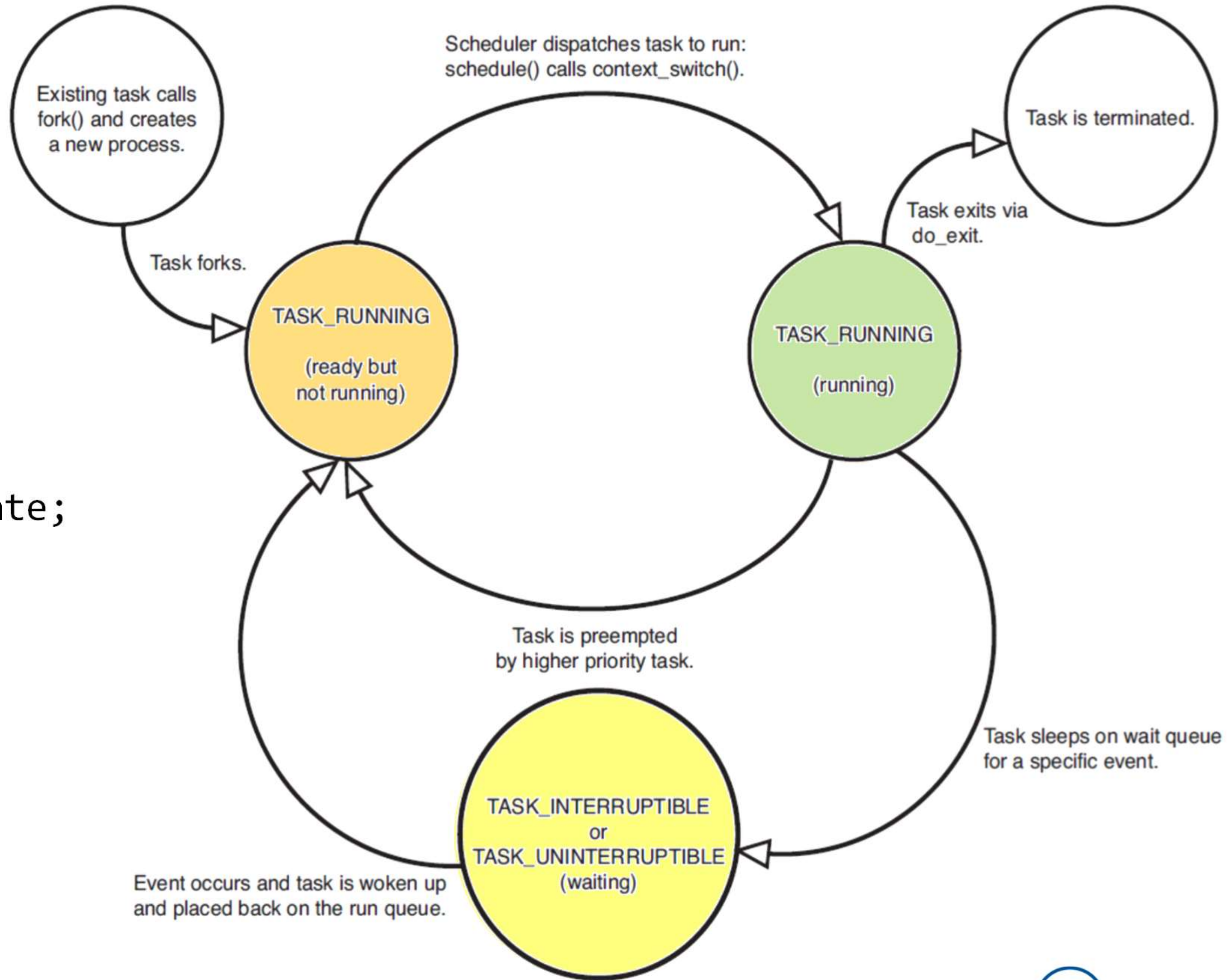
Reader-Writer Semaphores

```
static DECLARE_RWSEM(mr_rwsem);
//struct rw_semaphore mr_rwsem;
//init_rwsem(&mr_rwsem);

// attempt to acquire the semaphore for reading ...
down_read(&mr_rwsem);

// critical region (read only) ...

// release the semaphore
up_read(&mr_rwsem);

...

// attempt to acquire the semaphore for writing ...
down_write(&mr_rwsem);

// critical region (read and write) ...

// release the semaphore
up_write(&mr_rwsem);
```

SUNY Korea

Sleeping and Waking up

```c
struct task_struct {
    ...
    volatile long state;
    ...
};
```



SUNY Korea

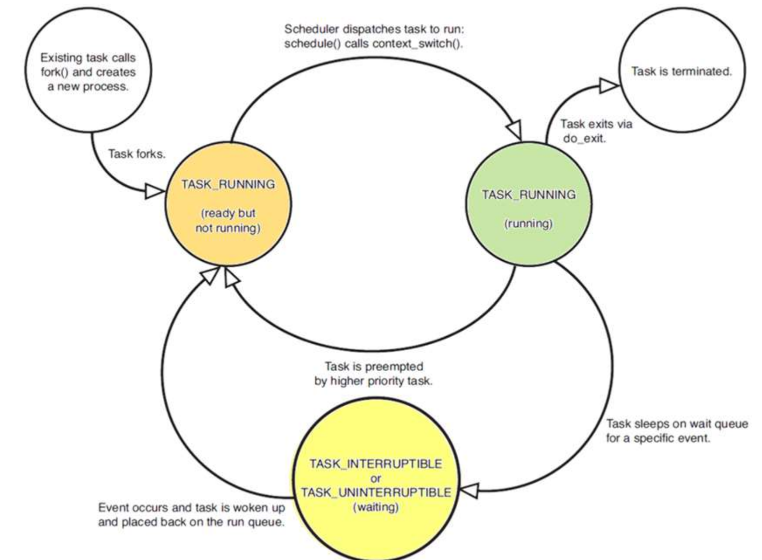
Sleeping and Waking up

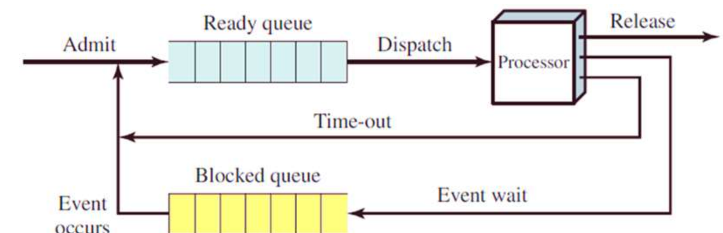
- Tasks that are sleeping (blocked) are in a special non-runnable state.
  - Scheduler would not select those tasks
  - All sleeping tasks are waiting for events
    - E.g. more data from file, h/w events, obtain a contended semaphore

SUNY Korea
The State University of New York

Sleeping and Waking up

- Kernel behaviors to sleep
  - Mark it’s state as sleeping:
    TASK_INTERRUPTABLE or
    TASK_UNINTERRUPTABLE
  - Put itself to a wait queue
  - Call schedule()
    - Remove the current process from the runqueue
    - Select a new process to run

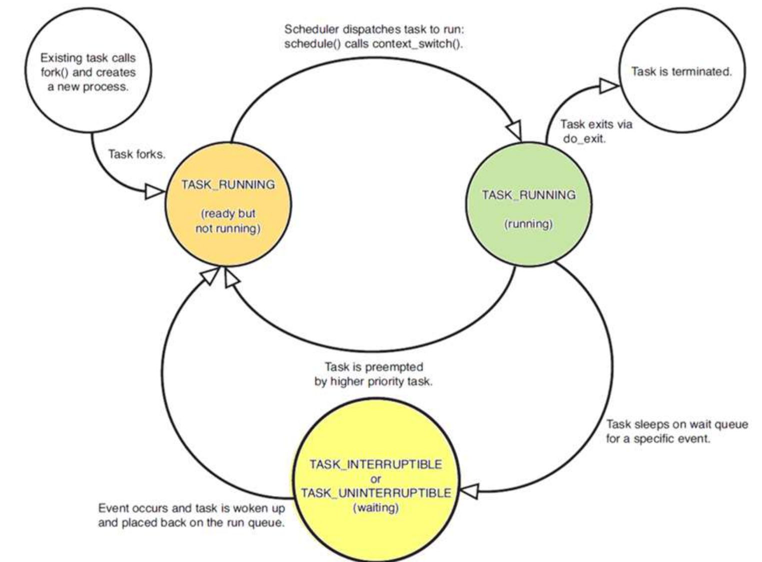


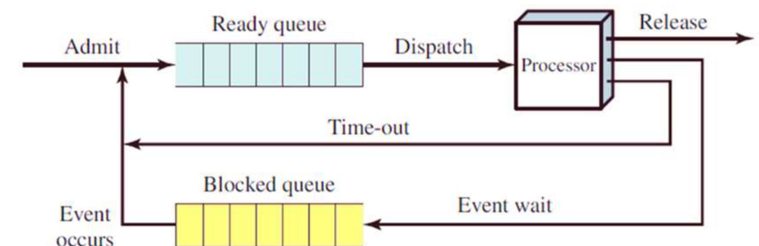


SUNY Korea

Sleeping and Waking up

- Kernel behaviors to wakeup
  - Set the task as runnable: TASK_RUNNING
  - Remove the task from the waitqueue
  - Add the task back to the runqueue





SUNY Korea

Sleeping and Waking up

- Wait queues
  - A wait queue is a simple list of processes waiting for an event to occur

```
//Each wait queue is identified by __wait_queue_head
struct __wait_queue_head {
    spinlock_t          lock;
    struct list_head    task_list;
};

//Elements of a wait queue is __wait_queue
struct __wait_queue {
    unsigned int        flags;
    struct task_struct  *task;
    wait_queue_func_t   func; //may set to default_wake_function
    struct list_head    task_list;
};
```

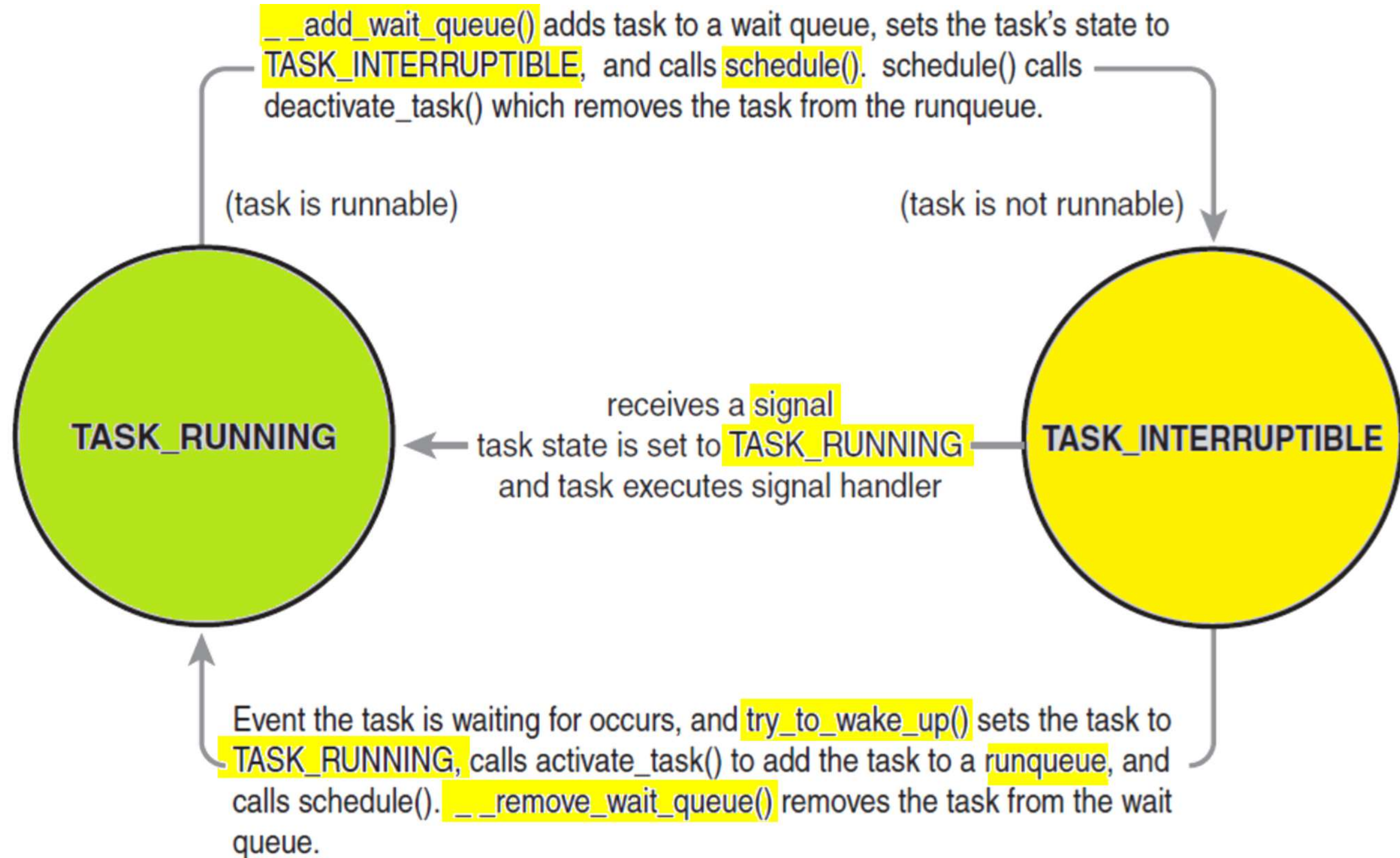
SUNY Korea

Sleeping and Waking up

- Waking up is handled by wake_up(), which calls try_to_wake_up()
  - Sets the task's state to TASK_RUNNING
  - Add the task to the runqueue
  - Set need_resched if the awakened task's priority is higher than the current task

SUNY Korea

Sleeping and Waking up



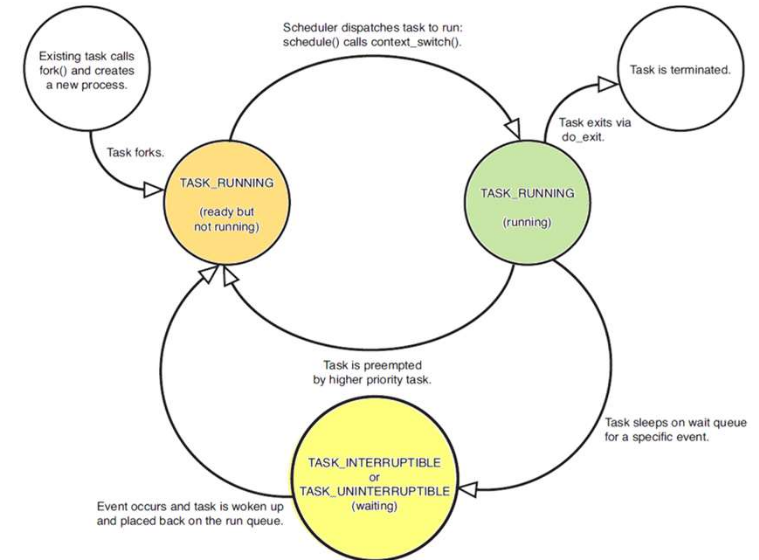
SUNY Korea

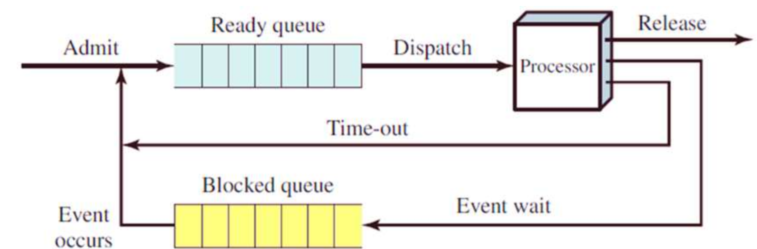
Event Example

```c
// event_eg.c
//
#include <linux/syscalls.h>
#include <linux/wait.h>
#include "common.h"


// wait queue
static DECLARE_WAIT_QUEUE_HEAD(wait_q);

SYSCALL_DEFINE0(event_sig_eg)
{
    wake_up(&wait_q);
    return 0;
}
```

SUNY Korea

```
SYSCALL_DEFINE0(event_wait_eg)
{
    DEFINE_WAIT(wait);
    printk("entering %s\n", __FUNCTION__);
#if 0
    init_waitqueue_entry(&wait, current);
    current->state = TASK_INTERRUPTIBLE;
    add_wait_queue_exclusive(&wait_q, &wait);
#else
    prepare_to_wait_exclusive(
        &wait_q, &wait, TASK_INTERRUPTIBLE);
#endif
    printk("before schedule\n");
    schedule(); //context switch
    printk("after  schedule\n");

#if 0
    remove_wait_queue(&wait_q, &wait);
#else
    finish_wait(&wait_q, &wait);
#endif
    printk("leaving  %s\n", __FUNCTION__);
    return 0;
}
```





SUNY Korea

```c
// event_wait_eg.c, user space program
//
#include <stdio.h>
#include <unistd.h>
#include "wrapper.h"

int main() {
    int i;
    long res;
    for (i = 0; i < 5; i++) {
        pid_t pid;
        if ((pid = fork()) == 0) {
            printf("[%d]: calling event_wait_eg\n", i);
            res = event_wait_eg();
            if (res)
                printf("event_wait_eg %d: res: %ld\n", i, res);
            printf("[%d]: return from evernt_wait_eg\n", i);
            return 0;
        }
    }
    sleep(1);
}
```

```c
//event_sig_eg.c, user space program
//

#include <stdio.h>
#include <unistd.h>
#include "wrapper.h"

int main() {
    long res;
    printf("calling event_sig_eg\n");
    res = event_sig_eg();
    if (res)
        printf("event_sig_eg res: %ld\n", res);
    printf("return from evernt_sig_eg\n");
}
```

```
$gcc -o ew event_wait_eg.c
$gcc -o es event_sig_eg.c
```

run ew and then run es 6 times

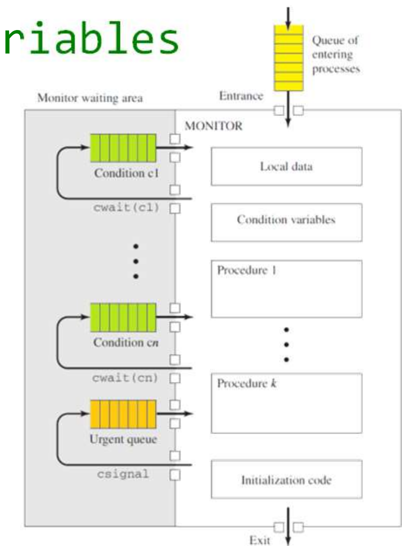
SUNY Korea

Take Home Midterm Exam

- In this exam, we will implement a monitor-like mechanism using the following 6 system calls
  - mon_create is a system call that creates a monitor
  - mon_destroy is a system call that destroys a monitor
  - mon_wait either admits a process to enter a monitor or makes it wait on a condition variable
  - mon_signal either makes a process leave a monitor or signals a condition variable
  - mon_notify notifies a conditional variable
  - mon_broadcast broadcasts to a conditional variable
- Due date TBD

```c
typedef struct monitor_struct {
    spinlock_t lock;                    //access lock
    int id;                             //id of the monitor
    int in_use;                         //whether this monitor is in use
    int nr_cv;                          //number of condition variables
    struct wait_queue_head *cond;       //condition variables
    struct wait_queue_head entry;       //entry queue
    struct wait_queue_head urgent;      //urgent queue
    struct list_head list;              //next monitor
} monitor_t;
```

```c
typedef struct monitor_manager {
    spinlock_t lock;                    //access lock
    int freshid;                        //id for the next monitor
    struct list_head mon_list;          //linked list of monitors
} monitor_manager_t;

static monitor_manager_t mon_mgr = {
    .lock = __SPIN_LOCK_UNLOCKED(lock),
    .freshid = 0,
    .mon_list = LIST_HEAD_INIT(mon_mgr.mon_list)
};
```



SUNY Korea
The State University of New York

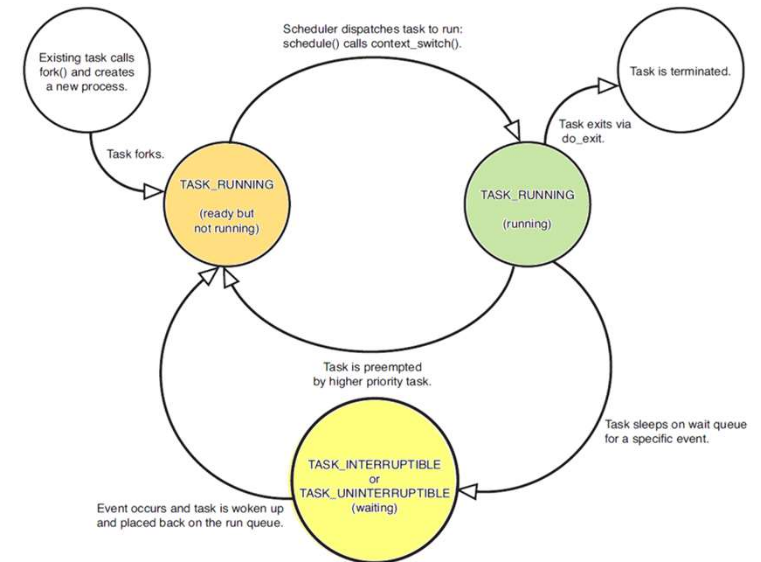
```c
//find the monitor with mon_id
static monitor_t *find_monitor(int mon_id) {
    monitor_t *ret = NULL, *mon;
    //TODO:
    // 1. lock mon_mgr.lock
    // 2. look for the monitor with mon_id in mon_mgr.mon_list
    // 3. unlock mon_mgr.lock
    // 4. return the monitor or NULL if not found
}

//create a new monitor
static monitor_t *create_monitor(int nr_cv) {
    monitor_t *ret = NULL, *mon = NULL;
    int i;
    mon = kmalloc(sizeof(monitor_t), GFP_KERNEL);
    ONFALSEGOTO(mon != NULL, ret = NULL, err);

    mon->lock = __SPIN_LOCK_UNLOCKED(lock),
    mon->in_use = 0;
    mon->nr_cv = nr_cv;
    mon->cond = kmalloc(sizeof(struct wait_queue_head) * nr_cv, GFP_KERNEL);
    ONFALSEGOTO(mon->cond != NULL, ret = NULL, err);
…
```

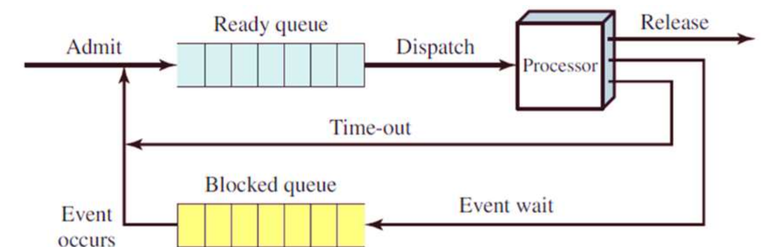
```
…
    //TODO: initialize wait queues: each of cond[i], entry and urgent


    //TODO:
    //1. lock mon_mgr.lock
    //2. set mon->id to mon_mgr.freshid and increase freshid
    //3. INIT_LIST_HEAD mon->list
    //4. add mon to mon_mgr.mon_list
    //5. unlock mon_mgr.lock

    return mon;
err:
    if(mon != NULL && mon->cond != NULL)
        kfree(mon->cond);
    if(mon != NULL)
        kfree(mon);
    return ret;
}
```

SUNY Korea

```
//destroy monitor
static void destroy_monitor(monitor_t *mon) {
    if(mon != NULL) {
        //TODO:
        //1. lock mon_mgr.lock
        //2. lock mon->lock
        //3. remove mon from mon_mgr.mon_list
        //4. unlock mon->lock
        //5. unlock mon_mgr.lock

        kfree(mon->cond);
        kfree(mon);
    }
}
```

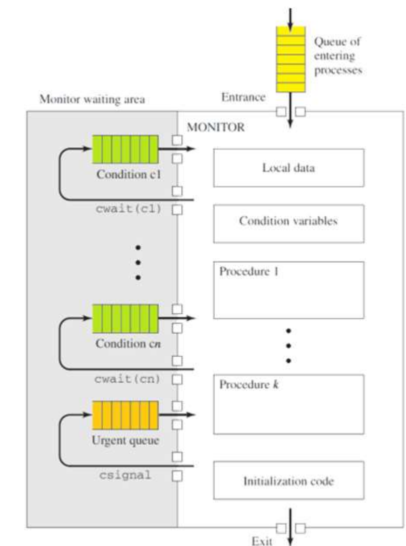
```
//wait on wqh
static void waiton(monitor_t *mon, struct wait_queue_head *wqh) {
    DEFINE_WAIT(wait);

    //TODO: using prepare_to_wait_exclusive,
    //1. add current to wait
    //2. add wait to wqh
    //3. set the current's state to TASK_INTERRUPTIBLE.


    //TODO:
    //1. unlock mon->lock
    //2. schedule (yield to other processes)
    //3. lock mon->lock


    //TODO: using finish_wait
    //remove wait from wqh
}
```





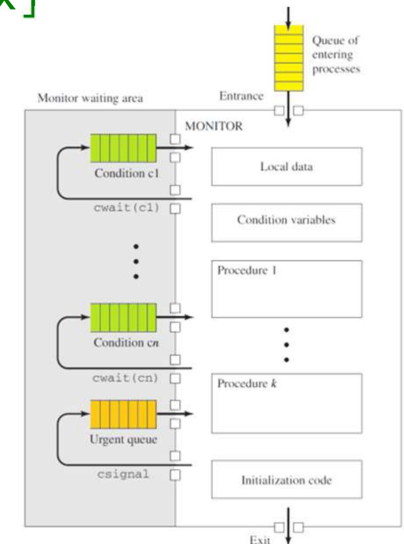
SUNY Korea

```c
//wakeup urgent queue or entry queue
//return 1 if any task is awaken; 0 otherwise
static int wakeup_urgent_or_entry(monitor_t *mon) {
    int ret = 1;
    //TODO: use wake_up_interruptible_sync instead of wake_up
    //wake_up_interruptible_sync lets the current running without
    //context switching
    //1. if mon->urgent is not empty wake up mon->urgent
    //2. else if mon->entry is not empty wake up mon->entry
    //3. else ret = 0


    return ret;
}

//wait on cond[cv_index]
static void cond_wait(monitor_t *mon, int cv_index) {
    //TODO: if wakeup_urgent_or_entry returns 0, set mon->in_use to 0

    //TODO: wait on mon->cond[cv_index]
}
```



SUNY Korea

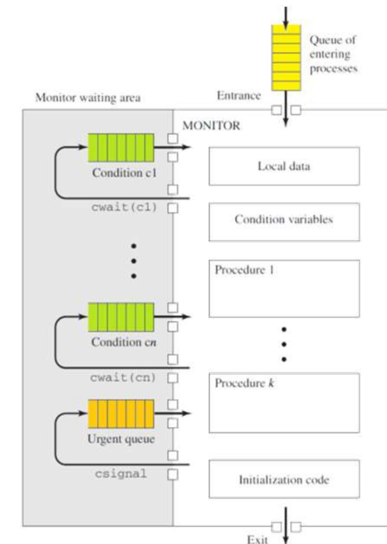
```
//signal to cond[cv_index]
static void cond_signal(monitor_t *mon, int cv_index) {
    //TODO: use wake_up_interruptible_sync instead of wake_up
    //wake_up_interruptible_sync lets the current running without
    //context switching
    //1. if cond[cv_index] is not empty, wake up cond[cv_index]
    //2. and wait on urgent (add the current to urgent)

}
```




```
//notify cond[cv_index]
static void cond_notify(monitor_t *mon, int cv_index) {
    struct wait_queue_entry *pwait;
    //TODO: move the first element of cond[cv_index] to entry
    //1. if cond[cv_index] is not empty, set pwait to its first element
    //2. remove pwait from the list
    //3. add pwait to mon->entry using add_wait_queue_exclusive
    //   tasks added exclusively will be awaken one by one

}
```

SUNY Korea

```c
//broadcast to cond[cv_index]
static void cond_broadcast(monitor_t *mon, int cv_index) {
    struct list_head *head = &mon->cond[cv_index].head;
    struct list_head *pos, *tmp;
    struct wait_queue_entry *pwait;
    //TODO: move the all elements of cond[cv_index] to entry
    //1. for each element from head (use list_for_each_safe)
    //2. remove pos from the list
    //3. let pwait be the container of pos
    //4. add pwait to mon->entry using add_wait_queue_exclusive
    //   tasks added exclusively will be awaken one by one
}

//enter the monitor
static void enter(monitor_t *mon) {
    //TODO:
    //if mon->in_use, wait on mon->entry
    //else set mon->in_use to 1
}
//leave the monitor
static void leave(monitor_t *mon) {
    //TODO:
    //if wakeup_urgent_or_entry returns 0
    //set mon->in_use to 0

}
```



```c
//create a monitor
SYSCALL_DEFINE1(mon_create, int, nr_cv) {
    long ret = 0;
    monitor_t *mon = NULL;
    //TODO
    //1. create a monitor with nr_cv condition variables
    //2. if NULL is returned, set ret = -ENOMEM and got out
    //3. set ret to mon->id

out:
    return ret;
}

//destroy the monitor with mon_id
SYSCALL_DEFINE1(mon_destroy, int, mon_id) {
    long ret = 0;
    monitor_t *mon = NULL;
    //TODO
    //1. find the monitor with mon_id
    //2. if not found, set ret = -EINVAL and goto out
    //3. else destroy the monitor

out:
    return ret;
}
```

SUNY Korea
The State University of New York

```c
//wait on the monitor with mon_id
//if cv_index == -1, wait on entry
//else wait on cond[cv_index]
SYSCALL_DEFINE2(mon_wait, int, mon_id, int, cv_index) {
    long ret = 0;
    monitor_t *mon = NULL;
    //TODO
    //1. find the monitor with mon_id
    //2. return -EINVAL if not found
    //3. lock mon->lock
    //4. if cv_index == -1, enter
    //5. else if 0 <= cv_index < mon->nr_cv, wait on cond[cv_index]
    //6. else set ret = -EINVAL and goto out

out:
    if(mon)
        spin_unlock(&mon->lock);
    return ret;
}
```

SUNY Korea

```c
//signal to the monitor with mon_id
//if cv_index == -1, leave
//else signal to cond[cv_index]
SYSCALL_DEFINE2(mon_signal, int, mon_id, int, cv_index) {
    long ret = 0;
    monitor_t *mon = NULL;
    //TODO
    //1. find the monitor with mon_id
    //2. return -EINVAL if not found
    //3. lock mon->lock
    //4. if cv_index == -1, leave
    //5. else if 0 <= cv_index < mon->nr_cv, signal to cond[cv_index]
    //6. else set ret = -EINVAL and goto out

out:
    if(mon)
        spin_unlock(&mon->lock);
    return ret;
}
```

SUNY Korea

```c
//notify the monitor with mon_id
SYSCALL_DEFINE2(mon_notify, int, mon_id, int, cv_index) {
    long ret = 0;
    monitor_t *mon = NULL;
    //TODO
    //1. find the monitor with mon_id
    //2. return -EINVAL if not found
    //3. lock mon->lock
    //4. if 0 <= cv_index < mon->nr_cv, notify cond[cv_index]
    //5. else set ret = -EINVAL and goto out

out:
    if(mon)
        spin_unlock(&mon->lock);
    return ret;
}
```

SUNY Korea

```c
//broadcast to the monitor with mon_id
SYSCALL_DEFINE2(mon_broadcast, int, mon_id, int, cv_index) {
    long ret = 0;
    monitor_t *mon = NULL;
    //TODO
    //1. find the monitor with mon_id
    //2. return -EINVAL if not found
    //3. lock mon->lock
    //4. if 0 <= cv_index < mon->nr_cv, broadcast to cond[cv_index]
    //5. else set ret = -EINVAL and goto out

out:
    if(mon)
        spin_unlock(&mon->lock);
    return ret;
}
```

SUNY Korea

Producer Consumer Problem with Monitor

```
enum cond_var {
        NotFull, NotEmpty
};
typedef struct bounded_buffer {
        int buf[N];
        int nextin, nextout;
        int count;
        int mid;        //monitor id
} bbuf_t;

void bbuf_init(bbuf_t *bbuf) {
        int ret = 0;
        bbuf->nextin = 0;
        bbuf->nextout = 0;
        bbuf->count = 0;
        bbuf->mid = mon_create(2); //monitor with 2 condition vars
        ONFALSEGOTO(bbuf->mid >= 0, bbuf->mid, out);
        return;
out:
        exit(0);
}
```

SUNY Korea

```c
void bbuf_add(bbuf_t *bbuf, int x) {
        int ret = 0;
        //TODO: enter monitor
        while(bbuf->count == N) {
                //TODO: wait on NotFull
        }
        bbuf->buf[bbuf->nextin] = x;
        bbuf->nextin = (bbuf->nextin + 1) % N;
        bbuf->count++;
        //TODO: signal NotEmpty
        //TODO: leave monitor
        return;
out:
        exit(0);
}

void bbuf_remove(bbuf_t *bbuf, int *x) {
        int ret = 0;
        //TODO: enter monitor
        if(bbuf->count == 0) {
            //TODO: wait on NotEmpty
        }
        *x = bbuf->buf[bbuf->nextout];
        bbuf->nextout = (bbuf->nextout + 1) % N;
        bbuf->count--;
        //TODO: notify NotFull
        //TODO: leave monitor
        return;
out:
        exit(0);
}
```

SUNY Korea

```c
void test_enter_leave() {  //monitor test in user space
        int ret = 0, mid;
        pid_t pid;
        printf("testing %s...\n", __FUNCTION__);
        mid = mon_create(1);
        ONFALSEGOTO(mid >= 0, mid, out);
        printf("mon id: %d\n", mid);

        if((pid = fork()) == 0) {
                printf("child  before enter\n");
                ONFALSEGOTO(0 == (ret = mon_enter(mid)), ret, out);
                printf("child  after  enter\n");

                sleep(2);

                printf("child  before leave\n");
                ONFALSEGOTO(0 == (ret = mon_leave(mid)), ret, out);
                printf("child  after  leave\n");

                exit(0);
        }
        else {
                sleep(1);

                printf("parent before enter\n");
                ONFALSEGOTO(0 == (ret = mon_enter(mid)), ret, out);
                printf("parent after  enter\n");

                printf("parent before leave\n");
                ONFALSEGOTO(0 == (ret = mon_leave(mid)), ret, out);
                printf("parent after  leave\n");

                waitpid(pid, NULL, 0);
        }
out:
…
```

```
testing test_enter_leave...
mon id: 0
child  before enter
child  after  enter
parent before enter
child  before leave
child  after  leave
parent after  enter
parent before leave
parent after  leave
done testing test_enter_leave
```

SUNY Korea

```c
void test_wait_signal() {  //monitor test in user space
…
        mid = mon_create(1);
…
        if((pid = fork()) == 0) {
                printf("child  before enter\n");
                ONFALSEGOTO(0 == (ret = mon_enter(mid)), ret, out);
                printf("child  after  enter\n");

                printf("child  before wait on 0\n");
                ONFALSEGOTO(0 == (ret = mon_wait(mid, 0)), ret, out);
                printf("child  after  wait on 0\n");

                printf("child  before leave\n");
                ONFALSEGOTO(0 == (ret = mon_leave(mid)), ret, out);
                printf("child  after  leave\n");

                exit(0);
        }
        else {
                sleep(1);

                printf("parent before enter\n");
                ONFALSEGOTO(0 == (ret = mon_enter(mid)), ret, out);
                printf("parent after  enter\n");

                printf("parent before signal to 0\n");
                ONFALSEGOTO(0 == (ret = mon_signal(mid, 0)), ret, out);
                printf("parent after  signal to 0\n");

                printf("parent before leave\n");
                ONFALSEGOTO(0 == (ret = mon_leave(mid)), ret, out);
                printf("parent after  leave\n");

                waitpid(pid, NULL, 0);
        }
```

```
testing test_wait_signal...
mon id: 1
child  before enter
child  after  enter
child  before wait on 0
parent before enter
parent after  enter
parent before signal to 0
child  after  wait on 0
child  before leave
child  after  leave
parent after  signal to 0
parent before leave
parent after  leave
done testing test_wait_signal
```

SUNY Korea

```c
void test_wait_notify() {  //test in user space
…
        mid = mon_create(1);
…
        if((pid = fork()) == 0) {
                printf("child  before enter\n");
                ONFALSEGOTO(0 == (ret = mon_enter(mid)), ret, out);
                printf("child  after  enter\n");

                printf("child  before wait on 0\n");
                ONFALSEGOTO(0 == (ret = mon_wait(mid, 0)), ret, out);
                printf("child  after  wait on 0\n");

                printf("child  before leave\n");
                ONFALSEGOTO(0 == (ret = mon_leave(mid)), ret, out);
                printf("child  after  leave\n");

                exit(0);
        }
        else {
                sleep(1);

                printf("parent before enter\n");
                ONFALSEGOTO(0 == (ret = mon_enter(mid)), ret, out);
                printf("parent after  enter\n");

                printf("parent before notify to 0\n");
                ONFALSEGOTO(0 == (ret = mon_notify(mid, 0)), ret, out);
                printf("parent after  notify to 0\n");

                printf("parent before leave\n");
                ONFALSEGOTO(0 == (ret = mon_leave(mid)), ret, out);
                printf("parent after  leave\n");

                waitpid(pid, NULL, 0);
        }
```

```
testing test_wait_notify...
mon id: 2
child  before enter
child  after  enter
child  before wait on 0
parent before enter
parent after  enter
parent before notify to 0
parent after  notify to 0
parent before leave
child  after  wait on 0
child  before leave
child  after  leave
parent after  leave
done testing test_wait_notify
```

```c
void test_wait_broadcast() {  //monitor test in user space
…
    for (i = 0; i < 3; i++) {
        if ((pid = fork()) == 0) {
            printf("child %d before enter\n", i);
            ONFALSEGOTO(0 == (ret = mon_enter(mid)), ret, out);
            printf("child %d after  enter\n", i);

            printf("child %d before wait on 0\n", i);
            ONFALSEGOTO(0 == (ret = mon_wait(mid, 0)), ret, out);
            printf("child %d after  wait on 0\n", i);

            printf("child %d before leave\n", i);
            ONFALSEGOTO(0 == (ret = mon_leave(mid)), ret, out);
            printf("child %d after  leave\n", i);

            exit(0);
        }
        else
            pids[i] = pid;
    }

    sleep(1);

    printf("parent  before enter\n");
    ONFALSEGOTO(0 == (ret = mon_enter(mid)), ret, out);
    printf("parent  after  enter\n");

    printf("parent  before broadcast to 0\n");
    ONFALSEGOTO(0 == (ret = mon_broadcast(mid, 0)), ret, out);
    printf("parent  after  broadcast to 0\n");

    printf("parent  before leave\n");
    ONFALSEGOTO(0 == (ret = mon_leave(mid)), ret, out);
    printf("parent  after  leave\n");
…
```

```
testing test_wait_broadcast...
mon id: 3
child 2 before enter
child 2 after  enter
child 1 before enter
child 0 before enter
child 2 before wait on 0
child 1 after  enter
child 1 before wait on 0
child 0 after  enter
child 0 before wait on 0
parent  before enter
parent  after  enter
parent  before broadcast to 0
parent  after  broadcast to 0
parent  before leave
child 2 after  wait on 0
child 2 before leave
child 1 after  wait on 0
child 1 before leave
child 1 after  leave
child 0 after  wait on 0
child 0 before leave
child 0 after  leave
child 2 after  leave
parent  after  leave
done testing test_wait_broadcast
```

SUNY Korea