Condition Variables and Mutexes

Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)
Systems Software & Architecture Lab.
Seoul National University

Fall 2024

Condition Variables and Mutex

- Yet another synchronization construct
- Condition variables can be also used without monitors in conjunction with mutexes
- Think of a monitor as a language feature
  - Under the covers, compiler knows about monitors
  - Compiler inserts a mutex to control entry and exit of processes to the monitor's procedures
  - But can be done anywhere in procedure, at finer granularity
- With condition variables, the module methods may wait and signal on independent conditions

Condition Variables

- Provide a mechanism to wait for events
  - A condition variable (CV) is an explicit queue
  - Threads can put themselves on CV when some state of execution is not met

- Used with mutexes
  - A mutex is a ____________ lock: threads are blocked when it is held by another thread
  - A mutex ensures mutual exclusion for a critical section
  - Manipulating some condition related to a CV should be done inside the critical section

CV Operations

- `wait(cond_t *cv, mutex_t *mutex)`
  - Assumes `mutex` is held when `wait()` is called
  - Puts the caller to sleep and releases `mutex` (atomically)
  - When awoken, reacquires `mutex` before returning
- `signal(cond_t *cv)`
  - Wakes a single thread if there are threads waiting on `cv`
  - Unlike semaphores, `signal()` is lost if there is no thread waiting for it
  - __________ semantics: thread continues after sending `signal()`
- `broadcast(cond_t *cv)`
  - Wakes all waiting threads
  - If there are no waiting thread, just return doing nothing

Pthreads Interface

- Mutexes and CVs are supported in Pthreads

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  c = PTHREAD_COND_INITIALIZER;

void wait_example() {
  pthread_mutex_lock(&m);
  pthread_cond_wait(&c, &m);
  pthread_mutex_unlock(&m);
}

void signal_example() {
  pthread_mutex_lock(&m);
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}
```

Joining Threads: An Initial Attempt

```
mutex_t m = MUTEX_INITIALIZER;
cond_t c = COND_INITIALIZER;

void *child(void *arg) {
  thread_exit();
  return NULL;
}

int main(int argc, char *argv[]) {
  thread_t p;
  thread_create(&p, NULL, child, NULL);
  thread_join();
  return 0;
}
```

```
void thread_exit() {
  mutex_lock(&m);
  cond_signal(&c);
  mutex_unlock(&m);
}

void thread_join() {
  mutex_lock(&m);
  cond_wait(&c, &m);
  mutex_unlock(&m);
}
```

Joining Threads: Second Attempt

- Keep state in addition to CVs

```
mutex_t m = MUTEX_INITIALIZER;
cond_t c = COND_INITIALIZER;
int done = 0

void *child(void *arg) {
  thread_exit();
  return NULL;
}

int main(int argc, char *argv[]) {
  thread_t p;
  thread_create(&p, NULL, child, NULL);
  thread_join();
  return 0;
}
```

```
void thread_exit() {
  done = 1;
  cond_signal(&c);
}

void thread_join() {
  mutex_lock(&m);
  if (done == 0)
    cond_wait(&c, &m);
  mutex_unlock(&m);
}
```

Joining Threads: Third Attempt

- Always hold mutex while signaling

```
mutex_t m = MUTEX_INITIALIZER;
cond_t c = COND_INITIALIZER;
int done = 0

void *child(void *arg) {
  thread_exit();
  return NULL;
}

int main(int argc, char *argv[]) {
  thread_t p;
  thread_create(&p, NULL, child, NULL);
  thread_join();
  return 0;
}
```

```
void thread_exit() {
  mutex_lock(&m);
  done = 1;
  cond_signal(&c);
  mutex_unlock(&m);
}

void thread_join() {
  mutex_lock(&m);
  while (done == 0)
    cond_wait(&c, &m);
  mutex_unlock(&m);
}
```

Bounded Buffer with CVs/Mutexes

```c
mutex_t m;
cond_t notfull, notempty;
int in, out, count;

void produce(data) {
  mutex_lock(&m);
  while (count == N)
    cond_wait(&not_full, &m);

  buffer[in] = data;
  in = (in+1) % N;
  count++;

  cond_signal(&not_empty);
  mutex_unlock(&m);
}
```

```c
void consume(data) {
  mutex_lock(&m);
  while (count == 0)
    cond_wait(&not_empty, &m);

  data = buffer[out];
  out = (out+1) % N;
  count--;

  cond_signal(&not_full);
  mutex_unlock(&m);
}
```

Using Broadcast

- Covering condition: when the signaler has no idea on which thread should be woken up
- e.g., memory allocation:

```
mutex_t m;
cond_t c;
int bytesLeft = MAX_HEAP_SIZE;

void free(void *p, int size) {
  mutex_lock(&m);
  bytesLeft += size;
  cond_broadcast(&c);
  mutex_unlock(&m);
}
```

```
void *allocate (int size) {
  mutex_lock(&m);
  while (bytesLeft < size)
    cond_wait(&c, &m);

  void *ptr = ...;
  bytesLeft -= size;
  mutex_unlock(&m);
  return ptr;
}
```

Semaphores vs. Mutexes + CVs

- Both have same expressive power
- Implementing semaphores using mutexes and CVs:

```
typedef struct sema_t {
  int v;
  cond_t c;
  mutex_t m;
} sema_t;

void sema_init(sema_t *s, int v) {
  s->v = v;
  cond_init(&s->c);
  mutex_init(&s->m);
}
```

```
void sema_wait(sema_t *s) {
  mutex_lock(&s->m);
  while (s->v <= 0)
    cond_wait(&s->c, &s->m);
  s->v--;
  mutex_unlock(&s->m);
}
void sema_signal(sema_t *s) {
  mutex_lock(&s->m);
  s->v++;
  cond_signal(&s->c);
  mutex_unlock(&s->m);
}
```

Xv6: Sleeplock

```
struct sleeplock {
  uint locked;
  struct spinlock lk;
  char *name;
  int pid;
};

void initsleeplock(struct sleeplock *lk,
                   char *name) {
  initlock(&lk->lk, “sleep lock”);
  lk->name = name;
  lk->locked = 0;
  lk->pid = 0;
}
```

```
void acquiresleep(struct sleeplock *lk) {
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}

void releasesleep(struct sleeplock *lk) {
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

Xv6: Sleep & Wakeup

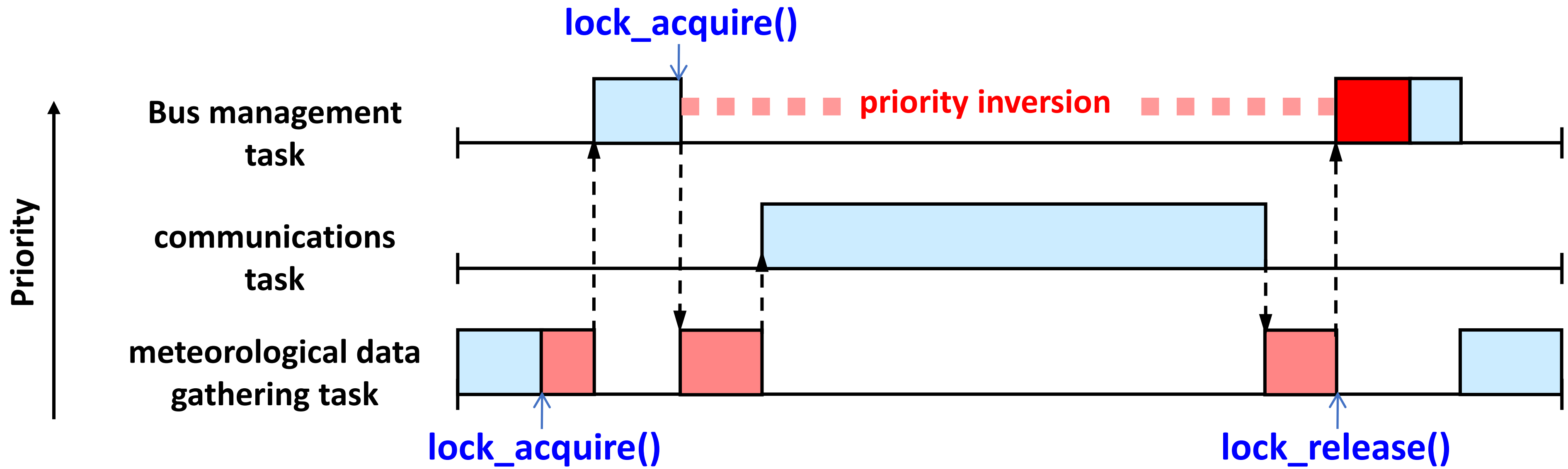
```
void sleep(void *chan,
           struct spinlock *lk) {
  struct proc *p = myproc();

  if (lk != &p->lock) {
    acquire(&p->lock);
    release(lk);
  }
  p->chan = chan;
  p->state = SLEEPING;
  sched();

  p->chan = 0;
  if (lk != &p->lock) {
    release(&p->lock);
    acquire(lk);
  }
}
```

```
void wakeup(void *chan) {
  struct proc *p;

  for (p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if (p->state == SLEEPING &&
        p->chan == chan) {
      p->state = RUNNABLE;
    }
    release(&p->lock);
  }
}
```

Concurrency Pitfalls

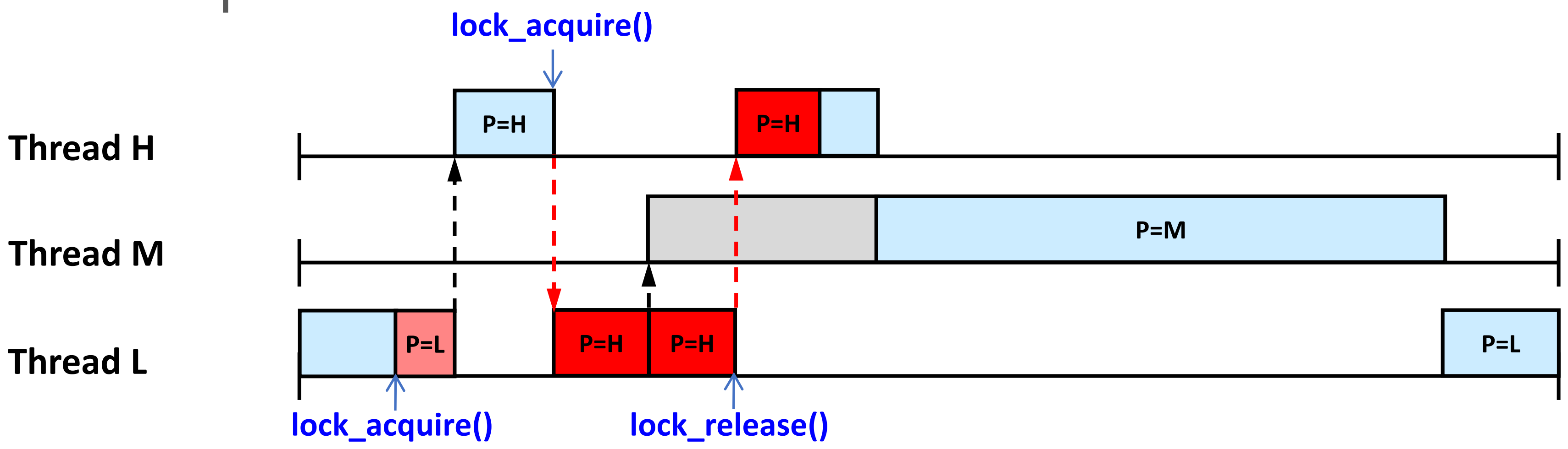
Priority Inversion

- Priority inversion problem
  - A situation where a higher-priority task is unable to run because a lower-priority task is holding a resource it needs, such as a lock
  - [What really happened on Mars?]



Priority Inversion: Solutions

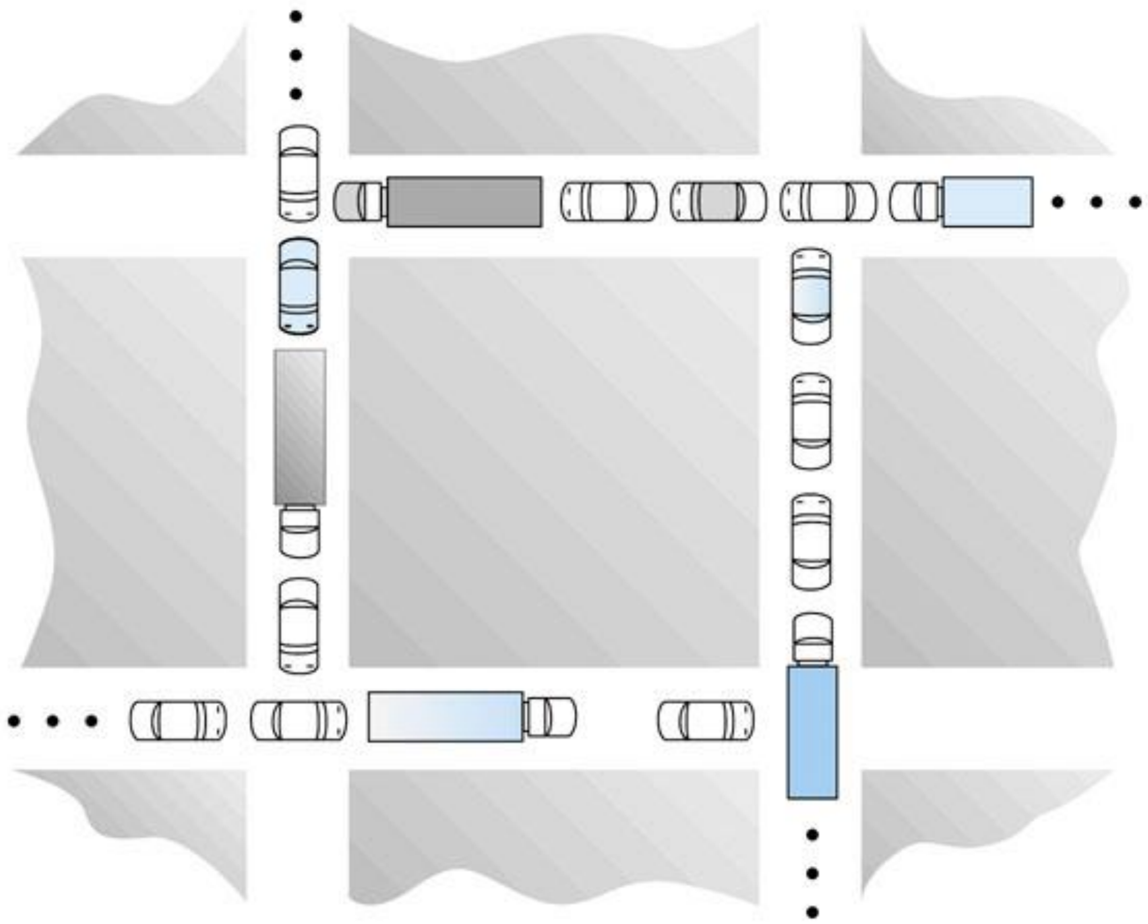
- Priority inheritance protocol (PIP)
  - The higher-priority task can donate its priority to the lower-priority task holding the resource it requires



- Priority ceiling protocol (PCP)
  - The priority of the low-priority task is raised immediately when it gets the resource
  - The priority ceiling value must be predetermined

Deadlock

- Traffic deadlock

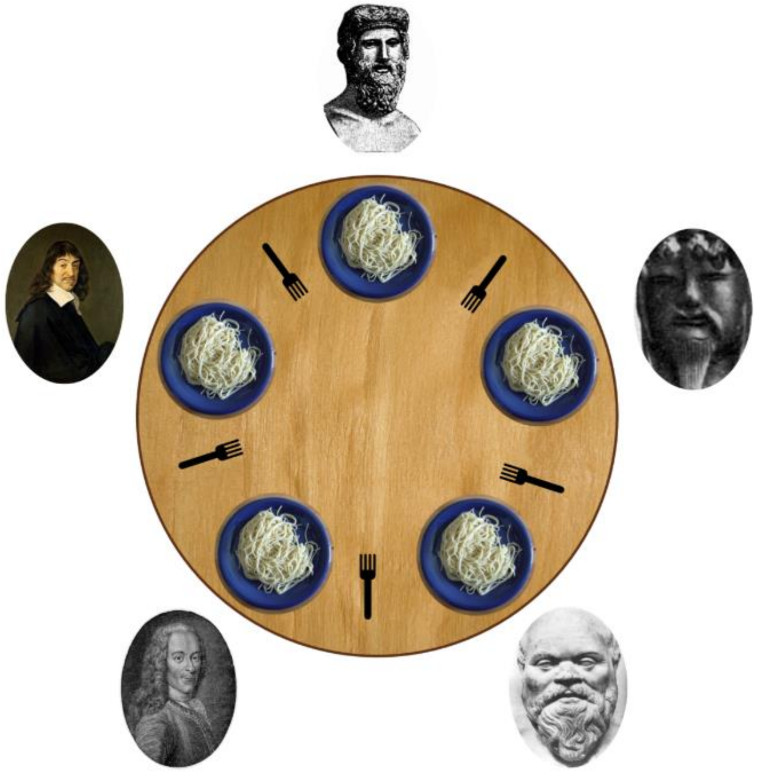


Deadlock: Examples

- Example 1

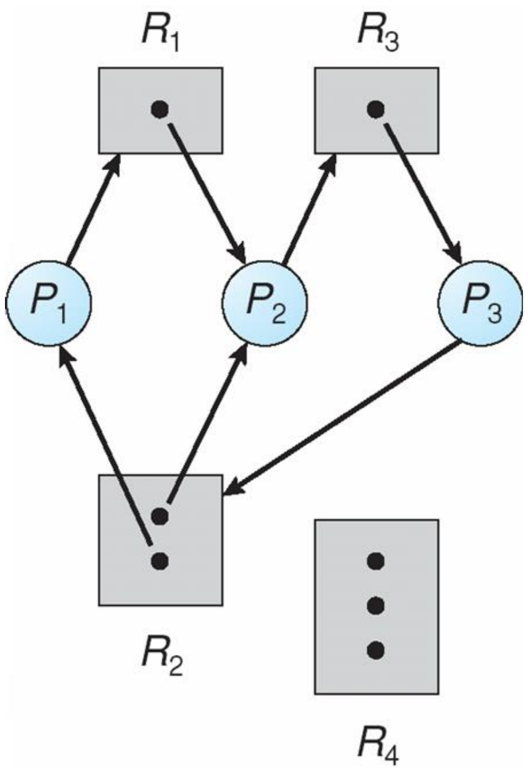
```
 void this() {
  acquire(&a);
  acquire(&b);
  // do this
  release(&b);
  release(&a);
}

void that() {
  acquire(&b);
  acquire(&a);
  // do that
  release(&a);
  release(&b);
}
```

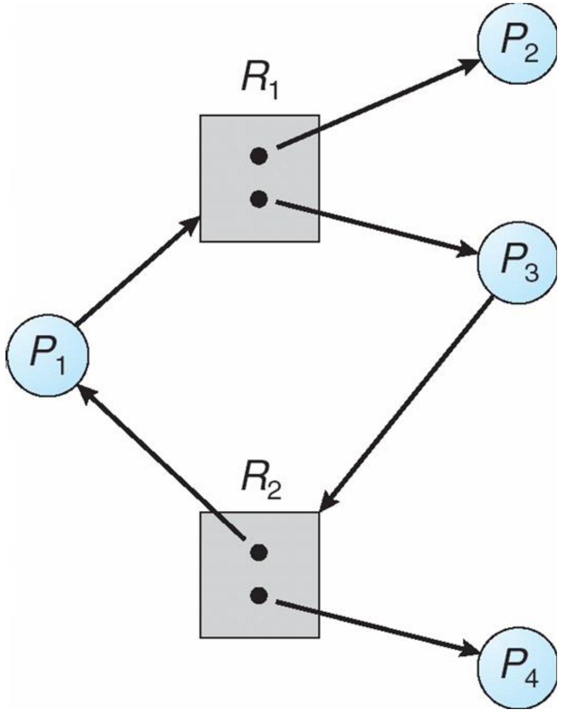
- Example 2



Deadlock Problem

- A set of blocked tasks each holding a resource and waiting to acquire a resource held by another process in the set





Source: A. Silberschatz et al., Operating System Concepts, 2008.

Necessary Conditions for Deadlock

- Mutual exclusion
  - Only one task at a time can use a resource
- Hold and wait
  - A task holding at least one resource is waiting to acquire additional resources held by other tasks
- No preemption
  - A resource can be released only voluntarily by the task holding it, after that task has completed its task
- Circular wait
  - There must exist a set $\{T_0, T_1, .., T_n, T_0\}$ of waiting tasks such that T_0 is waiting for a resource that is held by T_1, T_1 is waiting for a resource held by T_2, etc.

Handling Deadlocks

- Deadlock prevention
  - Restrain how requests are made
  - Ensure that at least one necessary condition cannot hold
- Deadlock avoidance
  - Require additional information about how resources are to be requested
  - Decide to approve or disapprove requests on the fly
- Deadlock detection and recovery
  - Allow the system to enter a deadlock state and then recover
- Just ignore the problem altogether!

Deadlock Prevention

- **Avoiding circular wait**
  - Impose a total ordering of all resource types, as a one-to-one function *F*.
    - $F: R \rightarrow N$, where $R = \{R_1, R_2, ..., R_n\}$ is the set of resource types and N is the set of natural numbers
    - e.g., *F*(lock a)=1, *F*(lock b)=2, *F*(lock c)=3, etc.
  - Each task requests resources in an increasing order of enumeration
  - Whenever a task requests an instance of R_j, it has released any resources R_i such that $F(R_i) >= F(R_j)$
  - *F* should be defined according to the normal order of usage of the resources in a system

```
 void this() {
  acquire(&a);
  acquire(&b);
  ...
  release(&b);
  release(&a);
}

void that() {
  acquire(&a);
  acquire(&b);
  acquire(&c);
  ...
  release(&c);
  release(&b);
  release(&a); }
```

Summary

- Disabling interrupts
  - Only for the kernel on a single CPU
- Spinlocks
  - Busy waiting, implemented using atomic instructions
- Semaphores
  - Binary semaphore = mutex (≅ lock)
  - Counting semaphore
- Monitors
  - Language construct with condition variables
- Mutexes + condition variables
  - Used in Pthreads