Operating Systems: Processes and Threads

keleher

February 20, 2024

Outline

1. Synchronization
2. Bounded counter

Locks

- Lock operations: acquire and release
- lck ← Lock()   // define a lock
- lck.acq()   // acquire the lock; blocking
  - call only if caller does not hold lck
  - returns only when no other thread holds lck
- lck.rel()   // release the lock; non-blocking
  - call only if caller holds lck
- lck.rel() does not give priority to threads blocked in lck.acq()

Condition variables

- Condition variable operations: wait, signal and signal_all
- A condition variable is associated with a lock
- cv ← Condition(lck)   // condition variable associated with lck
- cv.wait()   // wait on cv; blocking
  - call only if caller already holds lck
  - atomically release lck and wait on cv
    when awakened: acquire lck and return
- cv.signal()   // signal cv; non-blocking
  - call only if caller holds lck
  - wake up a thread (if any) waiting on cv
- cv.signal_all()   // wake up all threads waiting on cv
- lck.acq() does not give priority to threads blocked in cv.wait()

Why do conditionals have associated locks


```c
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

atomically release lock and wait()

re-acquires lock on wake

Figure 30.3: **Parent Waiting For Child: Use A Condition Variable**

Conditionals

- Two cases:

1. parent creates child, continues running:
   1.1 parent acquires lock
   1.2 checks for child done (no)
   1.3 go to sleep via `wait()`
   1.4 child eventually runs, exits
   1.5 parent wakes
2. child runs immediately
   2.1 child signals, exits
   2.2 parent wakes

Conditionals

- What if we didn't have the done state variable?

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Figure 30.4: **Parent Waiting: No State Variable**

- Fine if parent runs first....

Conditionals

- What if we didn't have the associated lock?

```
void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}

void thr_join() {
    if (done == 0)
        Pthread_cond_wait(&c);
}
```

Figure 30.5: **Parent Waiting: No Lock**

- Not good if parent check, then child runs....
- Referred to as a "TOCTOU" (Time_Of_Check to Time_Of_Use) vulnerability

Semaphores

- Semaphore: variable with a non-negative integer count
- Semaphore operations: P() and V()
- sem ← Semaphore(N) // define semaphore with count N (≥ 0)
- sem.P() // blocking
  - wait until sem.count > 0 then decrease sem.count by 1; return
  - checking sem.count > 0 and decrementing are one atomic step
- sem.V() // non-blocking
  - atomically increase sem.count by 1; return
- V() does not give priority to threads blocked in P()

Recall hypothetical *await*

- *await B*: *S*, where *S* is a code chunk (*no blocking or infinite loop*) and *B* is a boolean condition (*no side effects*):
  - execute *S* only if *B* holds, all in one atomic step
  - if *B* does not hold, wait
- *atomic S*: short for *await True*: *S*
- Example: Given a linked list x with non-blocking functions *add*() and *rmv*(). To allow multiple threads to call these functions simultaneously, simply wrap them as follows:
  - *await True* : *add*()
  - *await* (*xnotempty*) : *rmv*()

Progress assumption

- For a multi-threaded program to achieve anything, we have to assume that its threads execute with non-zero speed (but otherwise arbitrarily varying)
- Making this precise is simple for non-blocking statements but not for blocking statements (eg, acquire, wait, P, await)
- A thread at an non-blocking statement T eventually gets past T
  - Achieved if every unblocked thread periodically gets cpu cycles
- A thread at a blocking statement T eventually gets past T if T is *continuously* unblocked or *repeatedly* (but not continuously) unblocked
  - Achieved in most implementations only in a probabilistic sense, not in a deterministic sense

Outline

1. Synchronization
2. Bounded counter

Bounded counter

Program P0:

- x, y: global int variables; initially 0
- up(), down() // callable by multiple threads simultaneously
- up() increments x only if x < 100, and returns 2*x
- down() decrements x only if x > 0, and returns 2*x

- up():
```
int z
await (x < 100):
    x ← x+1
    z ← x
return 2*z
```

- down():
```
int z
await (x > 0):
    x ← x-1
    z ← x
return 2*z
```

Program P1:

- x, y // as in P0
- lck ← Lock()
- cvNF ← Condition(lck) // for guard (x < 100)
- cvNE ← Condition(lck) // for guard (x > 0)

- up():

```
int z
lck.acq()
while (not x < 100):
    cvNF.wait()
x ← x+1
z ← x
cvNE.signal()
lck.rel()
return 2*z
```

- down():

```
int z
lck.acq()
while (not x > 0):
    cvNE.wait()
x ← x-1
z ← x
cvNF.signal()
lck.rel()
return 2*z
```

Program P2:

- x, y // as in P0
- lck ← Lock()
- cv ← Condition(lck) // for both guards

- up():
```
int z
lck.acq()
while (not x < 100):
    cv.wait()
x ← x+1
z ← x
cv.signal_all()
lck.rel()
return 2*z
```

- down():
```
int z
lck.acq()
while (not x > 0):
    cv.wait()
x ← x-1
z ← x
cv.signal_all()
lck.rel()
return 2*z
```

P0 → semaphore program P3 (via P1)

Program P3:

- x, y   // as in P1
- mutex ← Semaphore(1)   // for lck
- gateNF ← Semaphore(0)   // for cvNF
- gateNE ← Semaphore(0)   // for cvNE

- up():

```
int z
mutex.P()
while (not x < 100)
    mutex.V()
    gateNF.P()
    mutex.P()
x ← x+1
z ← x
gateNE.V()
mutex.V()
return ← 2*z
```

- down():

```
int z
mutex.P()
while (not x > 0)
    mutex.V()
    gateNE.P()
    mutex.P()
x ← x-1
z ← x
gateNF.V()
mutex.V()
return ← 2*z
```