Operating Systems:
Processes and Threads

keleher

February 20, 2024
Outline

1. Synchronization
2. Bounded counter
Locks

- Lock operations: acquire and release

- \texttt{lck} \leftarrow \texttt{Lock()} \quad \text{// define a lock}

- \texttt{lck.acq()} \quad \text{// acquire the lock; blocking}
  - call only if caller does not hold \texttt{lck}
  - returns only when no other thread holds \texttt{lck}

- \texttt{lck.rel()} \quad \text{// release the lock; non-blocking}
  - call only if caller holds \texttt{lck}

- \texttt{lck.rel()} does not give priority to threads blocked in \texttt{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;
}
```

Figure 30.3: Parent Waiting For Child: Use A Condition Variable
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 \texttt{wait()}
   1.4 child eventually runs, exits
   1.5 parent wakes

2. child runs immediately
   2.1 child signals, exits
   2.2 parent wakes
What if we didn’t have the done state variable?

```c
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....
What if we didn’t have the associated lock?

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

```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 \( V() \)

\[
\begin{align*}
\text{sem} & \leftarrow \text{Semaphore}(N) \quad // \text{define semaphore with count } N \geq 0 \\
\text{sem}.P() & \quad // \text{blocking} \\
& \quad \text{wait until } \text{sem}.\text{count} > 0 \text{ then decrease } \text{sem}.\text{count} \text{ by 1; return} \\
& \quad \text{checking } \text{sem}.\text{count} > 0 \text{ and decrementing are one atomic step} \\
\text{sem}.V() & \quad // \text{non-blocking} \\
& \quad \text{atomically increase } \text{sem}.\text{count} \text{ by 1; return} \\
\text{V()} & \text{does not give priority to threads blocked in } P()
\end{align*}
\]
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 (e.g., acquire, wait, P, await)

- A thread at a 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
1. Synchronization
2. 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`  
- `lck ← Lock()`  
- `cvNF ← Condition(lck)`  
- `cvNE ← Condition(lck)`  

- `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  
- lck ← Lock()  
- cv ← Condition(lck)  

// as in P0

// 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
Program P3:

- \(x, y\)  
- \text{mutex} \leftarrow \text{Semaphore}(1) \quad // \text{as in P1}  
- \text{gateNF} \leftarrow \text{Semaphore}(0) \quad // \text{for lck}  
- \text{gateNE} \leftarrow \text{Semaphore}(0) \quad // \text{for cvNF} \quad // \text{for cvNE}  

\text{up():}
\begin{align*}
\text{int } z \\
\text{mutex.P()} \\
\text{while (not } x < 100) \\
\quad \text{mutex.V()} \\
\quad \text{gateNF.P()} \\
\quad \text{mutex.P()} \\
x &\leftarrow x + 1 \\
z &\leftarrow x \\
gateNE.V() \\
\text{mutex.V()} \\
\text{return } \because 2z
\end{align*}

\text{down():}
\begin{align*}
\text{int } z \\
\text{mutex.P()} \\
\text{while (not } x > 0) \\
\quad \text{mutex.V()} \\
\quad \text{gateNE.P()} \\
\quad \text{mutex.P()} \\
x &\leftarrow x - 1 \\
z &\leftarrow x \\
gateNF.V() \\
\text{mutex.V()} \\
\text{return } \because 2z
\end{align*}