Operating Systems Engineering

Sleep & Wakeup
[chapter #5]

By Dan Tsafrir, 2016-04-13
in the previous lecture...

When critical section is short

FINE-GRAINED SYNCHRONIZATION
in this lecture…

When the critical section is long, need to allow for sleep-waiting

COARSE-GRAINED SYNCHRONIZATION
Big picture

◆ Multiple threads execute simultaneously
  ❖ Multicore

◆ They share memory & devices
  ❖ So they need to synchronize

◆ Previously, we covered
  ❖ Spinlocks

◆ Next
  ❖ How to wait for time scales too long for spinning
Example – thread #1 waits for threads #2 to terminate

- Spinning is out of the question in such scenarios
  - Wasteful

- Periodic polling is less wasteful (periodically check if thread #2 is alive, say, every T time units)
  - Cons:
    - Trades off latency for CPU cycles (≈ T/2 on avg.)
    - Still wastes cycles (not as bad as spinning, but still)
    - Wasted energy (prevents core from entering a power-saving state)
    - Harder and more challenging to program than…

- Event-based is better
  - T1 tells thread-manager it wants to sleep until T2 terminates
  - Challenge: sleep-waiting interacts with locks & context switching; in this lecture we will see how/why…
Agenda

- xv6’s “sequence coordination” (sleep-waiting)
  - OS literature has rich set of other suitable primitives

- By using
  - sleep & wakeup (p. 25–26)

- We will implement
  - pipewrite, piperead (pp. 60–61)
  - wait, exit (p. 23–24)
Producer-consumer queue: take #1

- **Goal**
  - Queue allows one process to send a nonzero pointer to another process

- **Assume**
  - There is only one sender and one receiver, and
  - They execute on different cores

- **Then**
  - This code is correct

- **Problem:** it’s wasteful

```c
struct pcq { void *ptr; /* initialized to 0 */ };
void send(struct pcq *q, void *p) {
    while(q->ptr != 0) ;
    q->ptr = p;
}
void* recv(struct pcq *q) {
    void *p;
    while((p = q->ptr) == 0) ;
    q->ptr = 0;
    return p;
}
```
Sleep semantics

- **Sleep( channel )**
  - Get a “channel” (aka “wait channel”)
    - Any 32bit number
  - Mark caller as “SLEEPING” on channel & put it to sleep
  - Yield core
  - More than one process can sleep on channel

- **Wakeup( channel )**
  - Mark all SLEEPING processes on channel as “RUNNABLE”
  - And wake them up
  - If no processes are waiting: wakeup does nothing

- **Channel is just an arbitrary number**
  - => Callers of sleep / wakeup must agree on it
Pseudo code

**sleep(chan):**
- `curproc->chan = chan`
- `curproc->state = SLEEPING`
- `sched() /* => yield & go to sleep*/`

**wakeup(chan):**
- `foreach p in proc[]:
  if (p->chan == chan) && (p->state == SLEEPING):
    p->state = RUNNABLE /*+ wake p*/`
Producer-consumer queue: take #2

(assumption that there’s only one producer and one consumer still holds)

```c
void* /* consumer */
recv(struct pcq *q) {
    void *p;
    if ( q->ptr == 0 )
        sleep(q); // until q≠0
    p = q->ptr;
    q->ptr = 0;
    wakeup(q);
    return p;
}

void /* producer */
send(struct pcq *q, void *p) {
    if ( q->ptr != 0 )
        sleep(q); // until q=0
    q->ptr = p;
    wakeup(q);
}
```

(assumption that there’s only one producer and one consumer still holds)
Problem of lost wakeup 😞

(further assume q->ptr==0 and the following depicts a schedule)

```c
void* /* consumer */
recv(struct pcq *q) {
    void *p;

    if( q->ptr == 0 ) // yes!
        // TOCTTOU...
        sleep(q); // entered

    p = q->ptr;
    q->ptr = 0;
    wakeup(q);
    return p;
}
```

```c
void /* producer */
send(struct pcq *q, void *p) {
    if( q->ptr != 0 ) // no!
        sleep(q); // skipped

    q->ptr = p;
    wakeup(q);
}
```
Problem of lost wakeup ☹

void /* consumer */
recv(struct pcq *q) {
    void *p;
    if (q->ptr == 0) // yes!
        sleep(q); // entered
    p = q->ptr;
    q->ptr = 0;
    wakeup(q);
    return p;
}

void /* producer */
send(struct pcq *q, void *p) {
    if (q->ptr != 0) // no!
        sleep(q); // skipped
    q->ptr = p;
    wakeup(q);
}

We need recv to be largely atomic
Problem of lost wakeup 😞

```c
void /* consumer */
recv(struct pcq *q) {
    void *p;
    acquire( &q->lock );
    if( q->ptr == 0 )
        sleep(q);
    // …
    release( &q->lock );
}
```

```c
void /* producer */
send(struct pcq *q, void *p) {
    acquire( &q->lock );
    if( q->ptr != 0 )
        sleep(q);
    //…
    release( &q->lock );
}
```

We can’t do this… (deadlock)
So the conclusion is…

- **Sleep must know about locks!**
- **Sleep should somehow**
  - 1) Atomically release the lock and put the process to sleep
  - 2) Reacquire the lock when waking up before it returns
  - => we’ll add the lock to the prototype of sleep

- **Let’s add a lock to the queue:**

```c
struct pcq {
    void *ptr;
    struct spinlock lock;
};
```
Producer-consumer queue: take #3

void * consumer */
recv(struct pcq *q) {

    void *p;
    acquire( &q->lock );
    if( q->ptr == 0 )
        sleep(q, &q->lock);
        // release…acquire
    p = q->ptr;
    q->ptr = 0;
    wakeup(q);
    release( &q->lock );
    return p;
}

void /* producer */
send(struct pcq *q, void *p) {

    acquire( &q->lock );
    if( q->ptr != 0 )
        sleep(q, &q->lock);
            // release…acquire
    q->ptr = p;
    wakeup(q);
    release( &q->lock );
}
Producer-consumer queue: take #3

```c
void /* consumer */
recv(struct pcq *q) {
    void *p;
    acquire( &q->lock );
    if( q->ptr == 0 )
        sleep(q, &q->lock);
    // release…acquire
    p = q->ptr;
    q->ptr = 0;
    wakeup(q);
    release( &q->lock );
    return p;
}

void /* producer */
send(struct pcq *q, void *p) {
    acquire( &q->lock );
    if( q->ptr != 0 )
        sleep(q, &q->lock);
    // release…acquire
    q->ptr = p;
    wakeup(q);
    release( &q->lock );
}
```

Would it work for multiple consumers and producers?
Producer-consumer queue: take #4 (multiple consumers-producers)

```c
void* /* consumer */
recv(struct pcq *q) {

    void *p;

    acquire( &q->lock );
    while( q->ptr == 0 )
        sleep(q, &q->lock);
    // release…acquire
    p = q->ptr;
    q->ptr = 0;
    wakeup(q);
    release( &q->lock );
    return p;
}
```

```c
void /* producer */
send(struct pcq *q, void *p) {

    acquire( &q->lock );
    while( q->ptr != 0 )
        sleep(q, &q->lock);
    // release…acquire
    q->ptr = p;
    wakeup(q);
    release( &q->lock );
}
```
XV6 CODE
sleep/wakeup

- **Read code**
  - wakeup (2614), wakeup1 (2603), sleep (2553)

- **Claim**
  - If following rules, sleep() & wakeup1() don’t miss each other
sleep/wakeup

- Read code
  - wakeup (2614), wakeup1 (2603), sleep (2553)

- Claim
  - If following rules, sleep() & wakeup1() don’t miss each other

- Proof
  - wakeup1() always runs while holding two locks:
    - (i) lk, and
    - (ii) ptable.lock
  - Whereas sleep() holds at least one (or both) until process actually sleeps
sleep/wakeup

- Sometimes multiple processes sleep on same channel
  - E.g., trying to read from same pipe
- Yet, a single call to `wakeup()` wakes them all
  - One of them will win (e.g., read the data on the pipe)
  - Others will find there’s nothing to do (read)
  - => For them, wakeup event was “spurious” (מלאכותי, מזויף)
- Therefore
  - sleep is always called inside a loop
  - Looping also helps if different uses of sleep/wakeup accidentally choose exact same channel
    - Simply means more spurious wakeups
Using sleep/wakeup #1 – pipes

- Waiting for what?
  - Reader: to have something to read
  - Writer: to have enough buffer space to write

- Code
  - struct pipe (6011), pipewrite (6080), piperead (6101)
Using sleep/wakeup #2 – wait/exit

- **Waiting for what?**
  - A parent waits for its child to terminate
  - E.g., shell running foreground processes

- **Recall**
  - When a child dies it turns into a zombie
  - DEALLOCATED only after it is wait( )ed for by its parent
  - If parent dies before child => init becomes the parent

- **Read code**
  - `wait (2403), exit (2354)`
  - `kill (2625, exit=itself; kill=another);` challenges:
    - killed process might be running on another core
    - killed process might be sleep-waiting for an event, holding some kernel resource