Lecture 20: Synchronization

CS 105

April 8, 2019

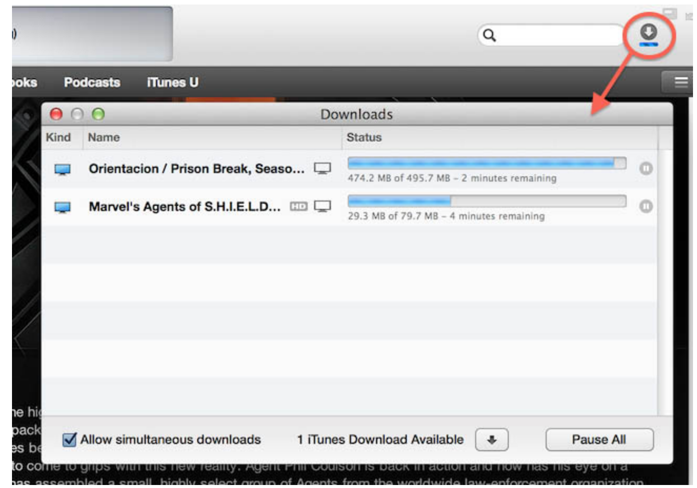
Last week: Processes and Threads

- A ***process*** is an instance of a running program.
  - logical control flow + isolated address space
- A ***thread*** is a sequential stream of execution
  - logical control flow + better performance

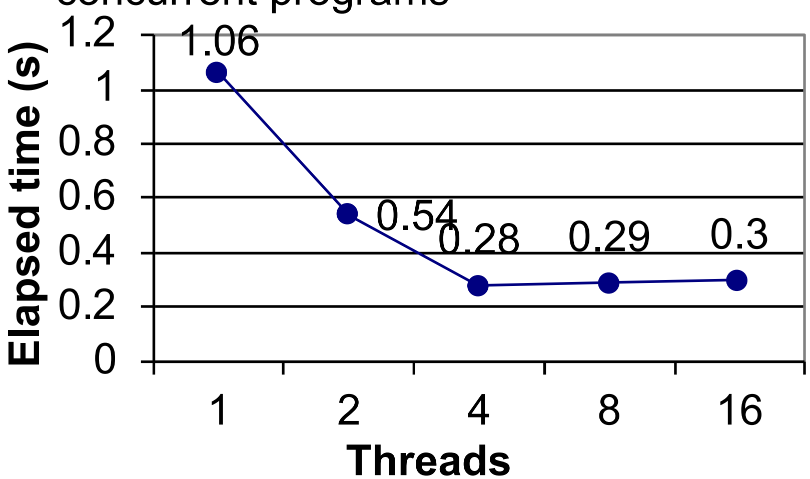
$\Rightarrow$ Concurrent Programs

Why Concurrent Programs?

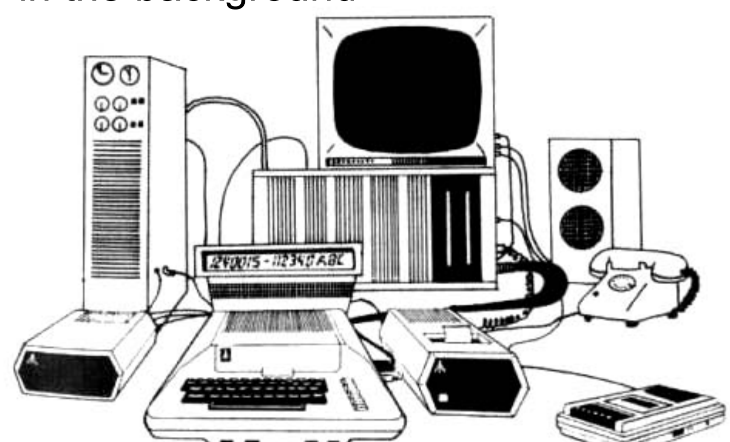
Program Structure: expressing logically concurrent programs



Responsiveness: shifting work to run in the background



Performance: exploiting multiprocessors



Performance: managing I/O devices

Why not Concurrent Programs?

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```c
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
                *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

```
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

Race condition!

Race conditions

- A race condition is a timing-dependent error involving shared state
  - whether the error occurs depends on thread schedule

- program execution/schedule can be non-deterministic
- compilers and processors can re-order instructions

A concrete example…

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.
- **Liveness:** if you are out of milk, someone buys milk
- **Safety:** you never have more than one quart of milk

Algorithm 1:

Look in fridge.
If out of milk:
    go to store,
    buy milk,
    go home
    put milk in fridge

A problematic schedule

You		Your Roommate	
3:00	Look in fridge; out of milk		
3:05	Leave for store		
3:10	Arrive at store	3:10	Look in fridge; out of milk
3:15	Buy milk	3:15	Leave for store
3:20	Arrive home; put milk in fridge	3:20	Arrive at store
		3:25	Buy milk
		3:30	Arrive home; put milk in fridge

Safety violation:

You have too much milk and it spoils

Solution 1: Leave a note

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Algorithm 2:

```
if (milk == 0) {      // no milk
  if (note == 0) {    // no note
    note = 1;         // leave note
    milk++;           // buy milk
    note = 0;         // remove note
  }
}
```

Safety violation: you've introduced a Heisenbug!

Solution 2: Leave note before check note

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Algorithm 3:

```
note1 = 1
if (note2 == 0) {    // no note fro
                        roommate
  if (milk == 0) {   // no milk
    milk++;          // buy milk
  }
}
note1 = 0
```

Liveness violation: No one buys milk

Solution 3: Keep checking for note

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Algorithm 4:

```
note1 = 1
while (note2 == 1) {// wait until
  ;                   //   no note
}
if (milk == 0) {      // no milk
  milk++;             // buy milk
}
note1 = 0
```

Liveness violation: you've introduced deadlock!

Solution 4: Take turns

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Algorithm 5:

```
note1 = 1
turn = 2
while (note2 == 1 and turn == 2){
  ;
}
if (milk == 0) {     // no milk
  milk++;            // buy milk
}
note1 = 0
```

(probably) correct, but complicated and inefficient

Rewind…

- What problem are we actually trying to solve?

Algorithm 1:

```
if (milk == 0) {     // no milk
  milk++;            // buy milk
}
```

- We want to limit the possible schedules so that checking for milk and buying milk act as a single **atomic** operation

Locks

- A **lock** (aka a mutex) is a synchronization that provides mutual exclusion. When one thread holds a lock, no other thread can hold it.
  - a lock can be in one of two states: locked or unlocked
  - a lock is initially unlocked

  - function acquire() waits until the lock is unlocked, then atomically sets it to locked
  - function release() sets the lock to unlocked

Atomic Operations

- Solution: hardware primitives to support synchronization
- A machine instruction that (atomically!) reads and updates a memory location

- Example: `xchg` *src, dest*
  - one instruction
  - semantics: TEMP ← DEST; DEST ← SRC; SRC ← TEMP;

Spinlocks

```
acquire:
    mov  $1,  eax          ; Set EAX to 1
    xchg eax, (rsi)        ; Atomically swap EAX w/ lock val
    test eax, eax          ; check if EAX is 0 (lock unlocked)
    jnz  acquire           ; if was locked, loop
    ret                    ; lock has been acquired, return

release:
    xor  eax, eax          ; Set EAX to 0
    xchg eax, (rsi)        ; Atomically swap EAX w/ lock val
    ret                    ; lock has been released, return
```

Solution 5: use a lock

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Algorithm 6:

```
acquire(&lock)
if (milk == 0) {      // no milk
  milk++;             // buy milk
}
release(&lock)
```

Correct!

Exercise: Dining Philosophers

```
eat_thread(i){
  while(True){
    think();

    pickup_fork(i);
    pickup_fork(i+1%n);

    eat();

    putdown_fork(i);
    putdown_fork(i+1%n);
  }
}
```

Locks in C (pthreads)

- Defines lock type pthread_mutex_t

- Defines functions to create/destroy locks:
  - int pthread_mutex_init(pthread_mutex_t *restrict *lock*, const pthread_mutexattr_t *restrict *attr*);
  - int pthread_mutex_destroy(pthread_mutex_t *mutex);

- Defines functions to acquire/release a lock:
  - int pthread_mutex_lock(pthread_mutex_t **lock*);
  - int pthread_mutex_trylock(pthread_mutex_t **lock*);
  - int pthread_mutex_unlock(pthread_mutex_t **lock*);

Exercise

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

```c
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
                 *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

- TODO: Modify this example to guarantee correctness

Performance problems

- threads that fail to acquire a lock on the first attempt must "spin", which wastes CPU cycles
  - replace no-op with yield()
- threads get scheduled and de-scheduled while the lock is still locked
  - need a better synchronization primitive

Better Synchronization Primitives

- Semaphores
  - stateful synchronization primitive

- Condition variables
  - event-based synchronization primitive

Semaphores

- A semaphore s is a stateful synchronization primitive comprised of:
  - a value (non-negative integer)
  - a lock
  - a queue

- Interface:
  - init(sem_t *s, 0, unsigned int val)
  - P(sem_t * s): If s is nonzero, the P decrements s and returns immediately. If s is zero, then adds the thread to queue(s); after restarting, the P operation decrements s and returns.
  - V(sem_t * s): Increments s by 1. If there are any threads in queue(s), then V restarts exactly one of these threads, which then completes the P operation.

Semantics of P and V

P():

- block (**sit on Q**) til value >0
- when so, decrement VALUE by 1

```
P() {
    while(n <= 0)
        ;
    n -= 1;
}
```

V():

- increment VALUE by 1
- **resume a thread waiting on Q (if any)**

```
V() {
    n += 1;
}
```

Binary Semaphore (aka mutex)

- A binary semaphore is a semaphore initialized with value 1.
  - the value is always 0 or 1
- Used for mutual exclusion---it's a more efficient lock!

```
Semaphore S
S.init(1)
```

T1

```
S.P()
CriticalSection()
S.V()
```

T2

```
S.P()
CriticalSection()
S.V()
```

Counting Semaphores

- Can also initialize semaphores with values greater than 1
- Can use these counting semaphores to do more complicated synchronization!

- … more on Wednesday