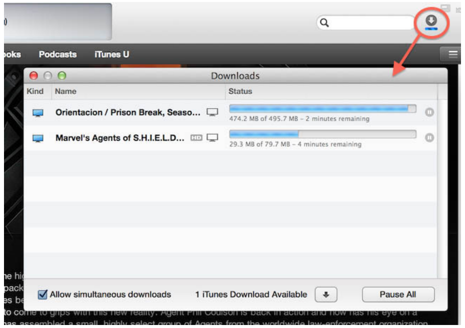
Lecture 21: Concurrency

CS 105

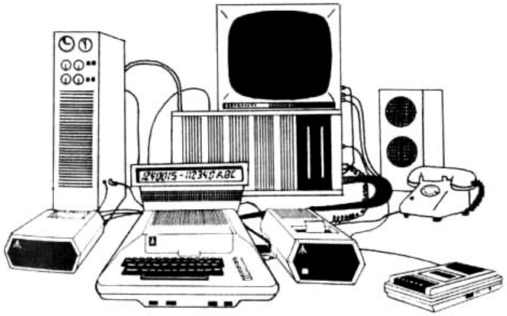
Why Concurrent Programs?

Program Structure: expressing logically concurrent programs

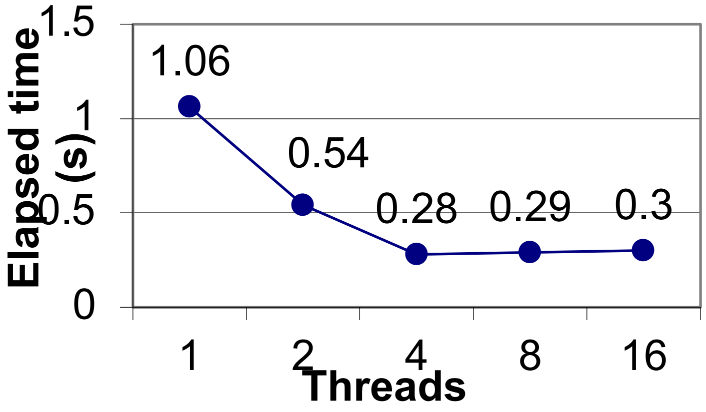
- Spell-check
- Autosaving
- Etc.



Responsiveness: shifting work to run in the background



Responsiveness: managing I/O devices



Performance: exploiting multiprocessors

Traditional View of a Process

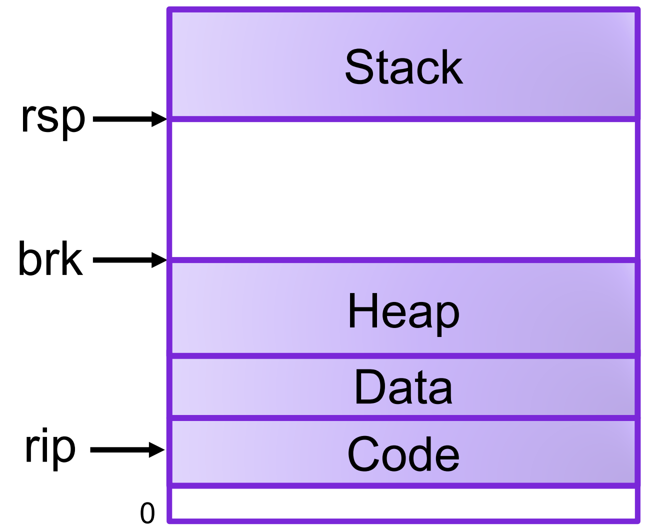
Process = process context + (virtual) memory state

Process Control Block

Program context:
- Data registers
- Stack pointer (rsp)
- Condition codes
- Program counter (rip)

Kernel context:
- VM structures
- File table
- brk pointer

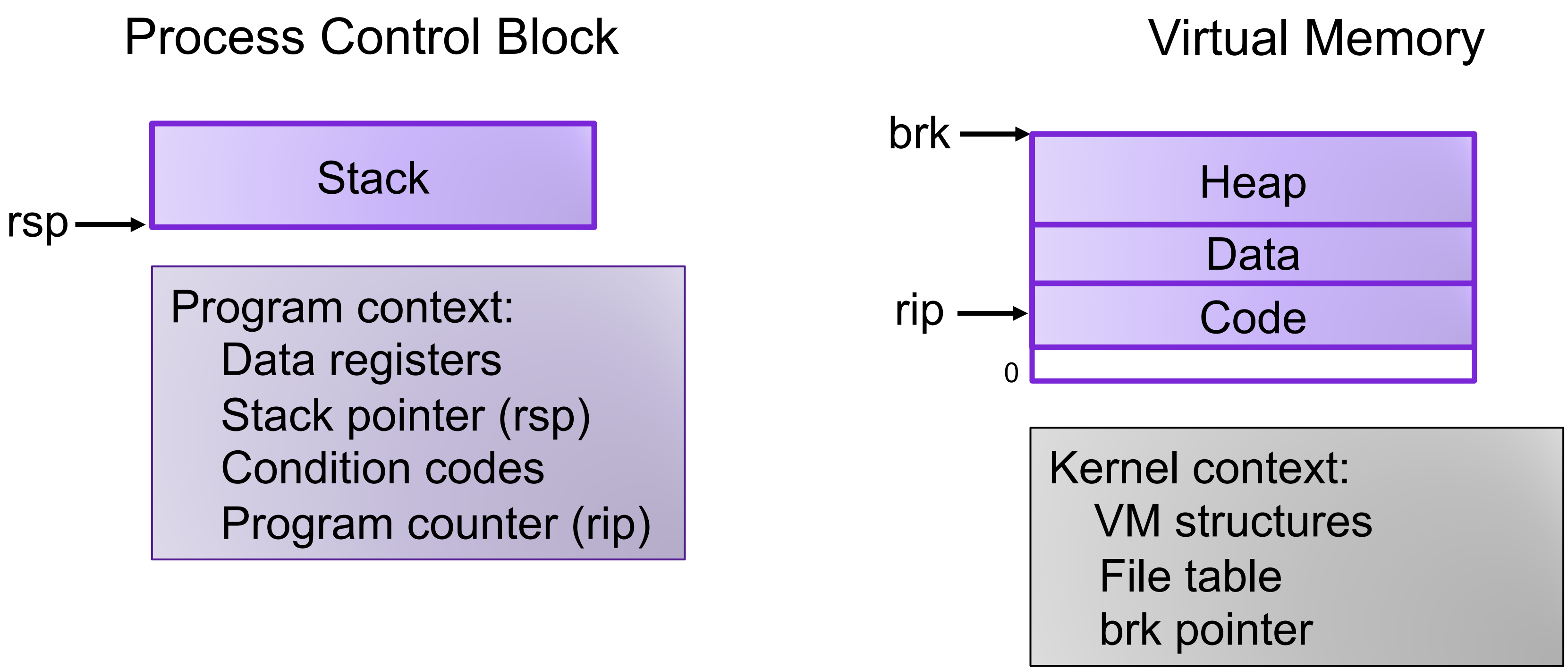
Virtual Memory



Alternate View of a Process

Process = ~~process context + (virtual) memory state~~

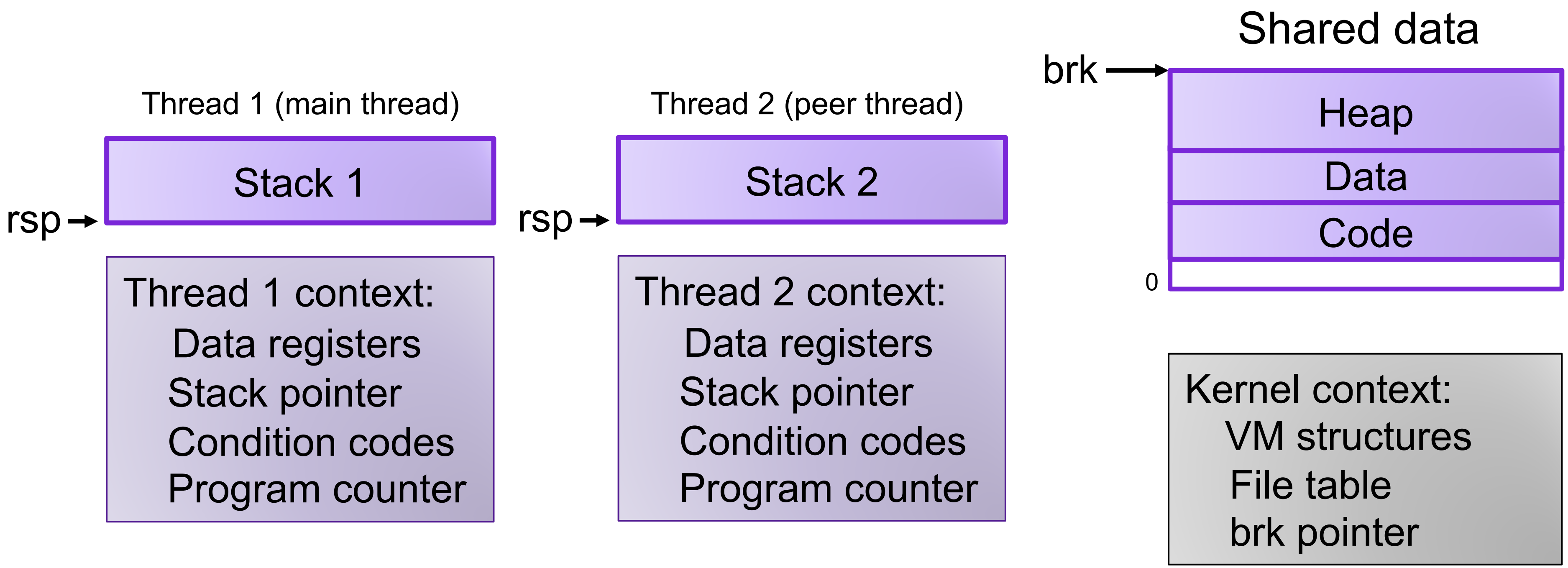
= thread context + process state



A Process With Multiple Threads

Multiple threads can be associated with a process

- Each thread has its own logical control flow
- Each thread has its own stack for local variables
- Each thread has its own thread id (TID)
- Each thread shares the same code, data, and kernel context



Threads vs. Processes

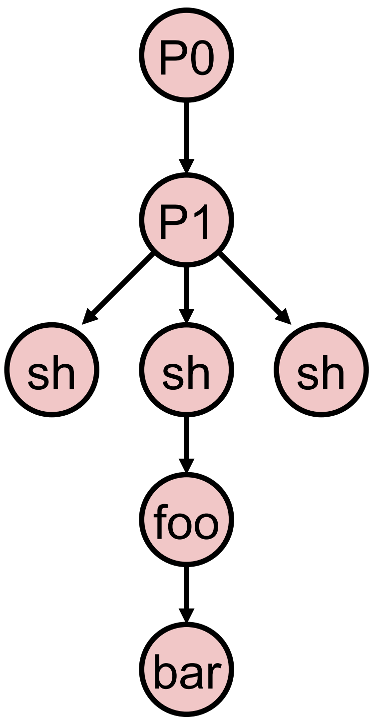
- How threads and processes are similar
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is scheduled and context switched

- How threads and processes are different
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Thread control (creating and reaping) is half as expensive as process control
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread
    - Thread context switches are less expensive (e.g., don't flush TLB)

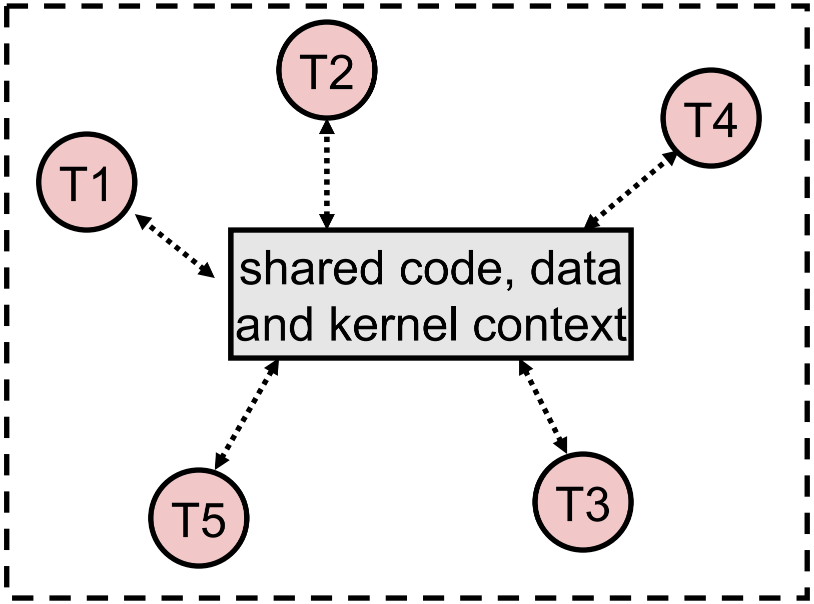
Logical View of Threads

- Threads associated with process form a pool of peers
- Unlike processes which form a tree hierarchy

Process hierarchy



Threads associated with process foo



Posix Threads Interface (Pthreads)

- Creating and reaping threads
  - `pthread_create()`
  - `pthread_join()`
- Determining your thread ID
  - `pthread_self()`
- Terminating threads
  - `pthread_cancel()`
  - `pthread_exit()`
  - `exit()` [terminates all threads]
  - `RET` [terminates current thread]

The Pthreads "hello, world" Program

```c
/*
 * hello.c - Pthreads "hello, world" program
 */

void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

hello.c

Thread ID

Thread attributes (usually NULL)

Thread routine

*Thread arguments (void *p)*

*Return value (void **p)*

Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

sharing.c

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
    return NULL;
}
```

Peer threads reference main thread's stack indirectly through global ptr variable

Mapping Variable Instances to Memory

- Global variables
  - *Def:* Variable declared outside of a function
  - **Virtual memory contains exactly one instance of any global variable**

- Local variables
  - *Def:* Variable declared inside function without `static` attribute
  - **Each stack frame contains one instance of each local variable**

- Local static variables
  - *Def:* Variable declared inside function with the `static` attribute
  - **Virtual memory contains exactly one instance of any local static variable.**

Mapping Variable Instances to Memory

```
char **ptr;  /* global var */

int main(){
  long i;
  pthread_t tid;
  char *msgs[2] = {"Hello from foo",
                   "Hello from bar"};
  ptr = msgs;
  for (int i = 0; i < 2; i++)
    Pthread_create(&tid, NULL,
                   thread, (void *)i);
  Pthread_exit(NULL);
}
```

```
void *thread(void *vargp){
  long myid = (long)vargp;
  static int cnt = 0;

  printf("[%ld]: %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
  return NULL;
}
```

Global var: 1 instance (`ptr` [data])

Local vars: 1 instance (`i.m`, `msgs.m`)

Local var: 2 instances (
  `myid.p0` [peer thread 0's stack],
  `myid.p1` [peer thread 1's stack]
)

Local static var: 1 instance (`cnt` [data])

Exercise 1: Shared Variables

```
char **ptr;  /* global var */

int main(){
  long i;
  pthread_t tid;
  char *msgs[2] = {"Hello from foo",
                   "Hello from bar"};
  ptr = msgs;
  for (int i = 0; i < 2; i++)
    Pthread_create(&tid, NULL,
                   thread, (void *)i);
  Pthread_exit(NULL);
}
```

```
void *thread(void *vargp){
  long myid = (long)vargp;
  static int cnt = 0;

  printf("[%ld]: %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
  return NULL;
}
```

Which variables are shared?

- `ptr`
- `cnt`
- `i.main`
- `msgs.main`
- `myid.thread0`
- `myid.thread1`

Exercise 1: Shared Variables

- Which variables are shared?
  - A variable `x` is shared iff multiple threads reference at least one instance of `x`.

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i.main`	yes	no	no
`msgs.main`	yes	yes	yes
`myid.thread0`	no	yes	no
`myid.thread1`	no	no	yes

Exercise 1: Shared Variables

- Which variables are shared?
  - A variable `x` is shared iff multiple threads reference at least one instance of `x`.

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i.main`	yes	no	no
`msgs.main`	yes	yes	yes
`myid.thread0`	no	yes	no
`myid.thread1`	no	no	yes

- **`ptr`, `cnt`, and `msgs` are shared**
- **`i` and `myid` are *not* shared**

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);
}
```

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

Assembly Code for Counter Loop

C code for counter loop in thread i

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

Asm code for thread i

```
    movq   (%rdi), %rcx
    testq  %rcx,%rcx
    jle    .L2
    movl   $0, %eax
.L3:
    movq   cnt(%rip),%rdx
    addq   $1, %rdx
    movq   %rdx, cnt(%rip)
    addq   $1, %rax
    cmpq   %rcx, %rax
    jne    .L3
.L2:
```

- H_i : Head (movq, testq, jle, movl)
- L_i : Load `cnt`
- U_i : Update `cnt`
- S_i : Store `cnt`
- T_i : Tail (addq, cmpq, jne)

Assembly Code for Counter Loop

C code for counter loop in thread i

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

Asm code for thread i

```
    movq   (%rdi), %rcx
    testq  %rcx,%rcx
    jle    .L2
    movl   $0, %eax
.L3:
    movq   cnt(%rip),%rdx
    addq   $1, %rdx
    movq   %rdx, cnt(%rip)
    addq   $1, %rax
    cmpq   %rcx, %rax
    jne    .L3
.L2:
```

H_i : Head

L_i : Load `cnt`

U_i : Update `cnt`

S_i : Store `cnt`

T_i : Tail

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:

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

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	3:20	Arrive at store
3:21	Put milk in fridge	3:25	Buy milk
		3:30	Arrive home
		3:31	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
  }
}
```

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 from
                     roommate
  if (milk == 0) {// no milk
    milk++;        // buy milk
  }
}
note1 = 0
```

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 from
                     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
```

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

Locks

- A **lock** (aka a mutex) is a synchronization primitive 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(&lock) waits until the lock is unlocked, then atomically sets it to locked
  - function release(&lock) sets the lock to unlocked

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

Correct!

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 $\leftarrow$ DEST; DEST $\leftarrow$ SRC; SRC $\leftarrow$ TEMP;

Spinlocks

```
acquire:
    mov  $1,  eax            ; Set EAX to 1
    xchg eax, (rdi)          ; 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, (rdi)          ; Atomically swap EAX w/ lock val
    ret                      ; lock has been released, return
```

Programming with Locks (Pthreads)

- Defines lock type pthread_mutex_t

- functions to create/destroy locks:
  - int pthread_mutex_init(&lock, *attr*)**;**
  - int pthread_mutex_destroy(&lock);

- functions to acquire/release lock:
  - int pthread_mutex_lock(&lock**);**
  - int pthread_mutex_unlock(&lock**);**

Exercise 2: Locks

```
/* 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);
}
```

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

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

    return NULL;
}
```

Modify this example to guarantee correctness

Problems with Locks

1. Locks are slow
   - Threads that fail to acquire a lock on the first attempt must "spin", which wastes CPU cycles
   - Threads get scheduled and de-scheduled while the lock is still locked

2. Using locks correctly is (surprisingly) hard
   - Hard to ensure all race conditions are eliminated
   - Easy to introduce synchronization bugs (deadlock, livelock)
   - Gets much harder when you have multiple needed resources

Better Synchronization Primitives

- Semaphores
  - stateful synchronization primitive
- Condition variables
  - event-based synchronization primitive