Example Threads

```
#include <pthread.h>
#include <stdio.h>

int num = 0;

void *add_one(int *thread_num) {
   num++;
   printf("thread %d  num = %d\n",
                  *thread_num, num);
}

void main() {
   pthread_t thread;
   int my_id = 0;
   int your_id = 1;
   pthread_create(&thread, NULL, add_one, &your_id)
   add_one(&my_id);
   pthread_join(thread, NULL);
   }
```

- compile: gcc mythread.cc -o mythread -lpthread
- What is the output of this program?

A Closer Look

```
sethi %hi(num),%o1
ld [%o1+%lo(num)],%o2
add %o2,1,%o1
st %o1,[%o0+%lo(num)]

sethi %hi(num),%o2
ld [%o1],%o1
ld [%o2+%lo(num)],%o2
call printf,0
```

- portion of the `add_one` assembly (obtained using gcc -S mythread.cc and looking at mythread.s)
- Timer interrupt can happen after any instruction (switching to another thread)
- What are the possible outputs?

The Critical Section Problem

```
while(1) {
   ...
   entry section //getting the lock
        critical section
   exit section // releasing the lock
  ... }
```

- Problem Description:
  - n processes competing to use shared data
  - Portions of the code that use the shared data are called *critical sections*
  - Problem: ensure only one process in the critical section
- An acceptable solution should:
  1. Ensure Mutual Exclusion (at most one process in the critical region)
  2. Ensure Progress is made (if region is empty, and there are processes that need it, they should be able to enter)
  3. Ensure no Starvation (after a process arrives, there is a bound on the number of processes that go in before it)

How to Implement Locks – Software Approaches

```
pthread_trylock(mutex) {
   if (mutex == 0) {
            mutex = 1;
            return 1;
   } else {
      return 0;
   }
}

Process 0, 1
.
.
while(!pthread_trylock(mutex));
<critical region>
pthread_unlock(mutex);
```

- Fictious implementation of trylock – does it work?
- What is the fundamental problem?

First Attempt: Better Solution

```
bool turn;

Process 0                   Process 1
.                           .
.                           .
while (turn != 0);          while (turn != 1);
[Critical Section]          [Critical Section]
turn = 1;                   turn = 0;
```

- Does this work?
- Which of the requirements are not satisfied?
- Drawbacks?
  - Strictly alternating order; may not map well to application needs
  - What if there is more than two?
  - What if a process fails?

Second Attempt: Separate Variables

```
bool flag[2];

Process 0                 Process 1
.                         .
.                         .
while (flag[1] != 0);     while (flag[0] != 0
flag[0] = 1;              flag[1] = 1;
[Critical Section]        [Critical Section]
flag[0] = 0;              flag[1] = 0;
```

- Problem Solved?
  - Strict turns do not have to be followed
  - Process failure still a problem?

- Is starvation a problem?

- Wrong Solution – why?

Third Attempt: Announce Interest Early

```
bool flag[2];

Process 0                    Process 1
.                            .
.                            .
flag[0] = 1;                 flag[1] = 1;
while (flag[1] != 0);        while (flag[0] != 0
[Critical Section]           [Critical Section]
flag[0] = 0;                 flag[1] = 0;
```

- Problem Solved?
  - Only one process can enter critical region at a time

- Is starvation a problem?

- Still a wrong Solution! why?

Fourth Attempt: Double check and Back-off

```
bool flag[2];
```

```
Process 0
.
.
flag[0] = 1;
while(flag[1] != 0) {
  flag[0] = 0;
  wait a short time
  flag[0] = 1;
}
[Critical Section]
flag[0] = 0;
```

```
Process 1
.
.
flag[1] = 1;
while(flag[0] !=
  flag[1] = 0;
  wait a short ti
  flag[1] = 1;
}
[Critical Section
flag[1] = 0;
```

- Finally a correct implementation?

Correct Alg.: Dekker's Algorithm

```
bool flag[2];
int turn = 0;
```

```
Process 0
...
flag[0] = 1;
while (flag[1] != 0) {
if (turn == 1) {
  flag[0] = 0;
  while (turn == 1);
  flag[0] = 1;
  } /*if*/
} /*while*/
[Critical Section]
flag[0] = 0;
turn = 1;
```

```
Process 1
...
flag[1] = 1;
while (flag[0] != 0) {
if(turn == 0) {
  flag[1] = 0;
  while (turn == 0);
  flag[1] = 1;
  } /*if*/
} /*while*/
[Critical Section]
flag[1] = 0;
turn = 0;
```

- The two flags solve the mutual exclusion problem; use the turn (as per the first implementation) to solve simultaneous interest problem
- Do we have the alternating execution problem?

More Elegant Solution: Peterson's Algorithm

```
bool flag[2];
int turn = 0;

Process 0                      Process 1
.                              .
.                              .
flag[0] = 1;                   flag[1] = 1;
turn = 1;                      turn = 0;
while (flag[1] == 1            while (flag[0] == 1
       && turn == 1);                 && turn == 0
[Critical Section]             [Critical Section]
flag[0] = 0;                   flag[1] = 0;
```

- Does this work? How?
- Is it fair (starvation/alternating execution?)
- How can we prove its correctness?

Bakery Algorithm

- Both Dekker's algorithm and Peterson's algorithm have generalizations for n processes (difficult; one will be a bonus homework question)
- Dijkstra's Bakery Algorithm also implements a critical section for n processes
- Idea: simulate operation in a bakery
  - Before entering the critical section (Bakery) receive a ticket number
  - The holder of the lowest ticket number gets in first
  - How do we ensure mutual exclusion on the ticket number? Cant two processes get the same ticket number?
    - $*$ Use the process id as a tie-breaker. If P_i and P_j have the same ticket number, and $i < j$, P_i gets in first

Bakery Algorithm

```
//choosing, ticket are shared
   ...
   choosing[i] = TRUE;
   ticket[i] = max (ticket[0], ticket [1] ...
                    ticket [n]) + 1;
   choosing[i] = FALSE;
   for(j = 0; j < n; j++) {
       while (choosing[j] == TRUE);
       while (ticket[j] != 0 &&
              (ticket[j],j) < (ticket [i],i));
     }
     [Critical Section]
    ticket[i] = 0;
    ...
```

- `(ticket[j],j) < (ticket[i],i)` refers to the comparison including using the process number as tie-breaker if tickets equal
- Take your time, think about it
- Does it satisfy the three requirements?

Hardware Mechanisms

- Software algorithms are difficult to understand and program
- Difficult to generalize (more than two processes, more than one lock)
- Inefficient
- Hardware mechanisms offer special atomic instructions that make building locks much easier
- Most of these instructions read a variable/change its value in one atomic operation
- Special Case: interrupt disabling for uniprocessors

Test and Set

- A single instruction that tests a boolean variable and sets it to 1 in one fell swoop (returns value before setting the variable)
- Atomicity guaranteed by the hardware
- Can something as simple as this help?
- Can we design a simpler (and preferrably correct :-) version of a lock using this instruction?

Test and Set Algorithm

```
bool lock = 0;

Process 0                      Process 1
.                              .
.                              .
while (testAndSet(lock)); while (testAndSet(lock));
[Critical Section]        [Critical Section]
lock = 0;                 lock = 0;
```

- Simpler
- Still busy waits
- Generalizes to any number of processes/locks
- What are the implications if used on a Shared Memory Multiprocessor?
- Is waiting bounded?
- Example of `test-and-op` class of primitives

Test and Set for n Processes with Bounded Wait

```
waiting[i] = 1;
key[i]=1;
while(waiting[i] && key[i])
    key[i] = testAndSet(lock);
waiting[i] = 0;

[Critical Section]

j = i+1 % n
while ((j != i) && !waiting[j])
     j = j + 1 % n;
if (j == i)
   lock = 0;
else
   waiting[j] = 0;
```

Busy waiting vs. Blocking

- All the methods discussed so far employ busy waiting
  - Such locks are called **spin locks**
    - A process waiting on a lock keeps spinning its wheels wasting CPU time
- Idea: use a blocking lock and signalling for a more efficient implementation – what is the tradeoff?
- Are there situations where spin locks are more efficient than blocking locks?
- Use locks as low-level primitives, but do not busy wait
- Semaphores (Dijkstra) is a widely used locking mechanism that uses this idea