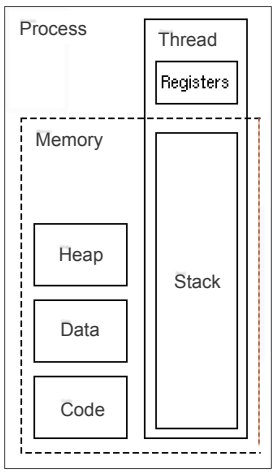
Computer Systems

Synchronizing Threads

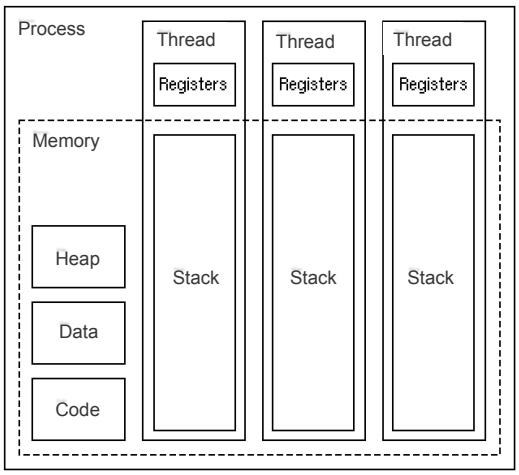
with

Semaphores

Review: Multi-Threaded Processes



single-threaded



multi-threaded

Review: `badcnt.c`

```
#define NITERS 1000000
```

```
unsigned int cnt = 0; /* shared */

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL,
                   Count, NULL);
    pthread_create(&tid2, NULL,
                   Count, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n",
                cnt);
    else
        printf("OK cnt=%d\n",
                cnt);
}
```

```
/* thread routine */
void * Count(void *arg) {
    int i;
    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}
```

```
linux> ./badcnt
BOOM! cnt=1988411

linux> ./badcnt
BOOM! cnt=1982618

linux> ./badcnt
BOOM! cnt=1982696
```

cnt should be equal to 2,000,000. What went wrong?!

Review: Race Conditions

- Assume that two threads execute concurrently

  `cnt++;` /* `cnt` is global shared data */

- One possible interleaving of statements is:

```
R1 = cnt
R1 = R1 + 1
Timer interrupt!
R2 = cnt
R2 = R2 + 1
cnt = R2
Timer interrupt!
cnt = R1
```

Race condition:

The situation where several threads access shared data concurrently. The final value of the shared data may vary from one execution to the next.

- Then `cnt` ends up incremented once only!

Review: Critical Sections

- Blocks of code that access shared data:

```
/* thread routine */
void * Count(void *arg) {
    int i;
    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}
```

- Threads need mutual exclusive access to critical sections: two threads cannot simultaneously execute cnt++

The Critical-Section Problem

- *Critical section (CS):* block of code that uses shared data
- *CS problem*: correctness of the program should not depend on one thread reaching a certain point before another thread
- *CS solution*: allow a *single thread* in the CS at one time

```
while(1)                                  Thread T_i
{  entry section
       Critical Section (CS)
   exit section
       Remainder Section (RS)
}
```

```
while(1) {
   entry section
      Critical Section (CS)
   exit section
      Remainder Section (RS)
}
```

Thread T_1

```
while(1) {
   entry section
      Critical Section (CS)
   exit section
      Remainder Section (RS)
}
```

Thread T_2

Solving the CS Problem - Semaphores (1)

- A *semaphore* is a synchronization tool provided by the operating system
- A semaphore S can be viewed as an integer variable that can be accessed through 2 *atomic* operations:

`DOWN(S)`	also known as	`wait(S)`
`UP(S)`	also known as	`signal(S)`

Atomic means *indivisible*.

- When a thread has to wait, put it in a *queue of blocked threads* waiting on the semaphore

Semaphores (2)

- In fact, a semaphore is a structure:

```
struct Semaphore {
        int count;
        Thread * queue; /*  blocked threads */
};
struct Semaphore S;     /* global variable */
```

- `S.count` must be initialized to a nonnegative value (depending on application)

OS Semaphores – `DOWN(S)` or `wait(S)`

- When a process must wait for a semaphore S, it is blocked and placed on the semaphore's queue

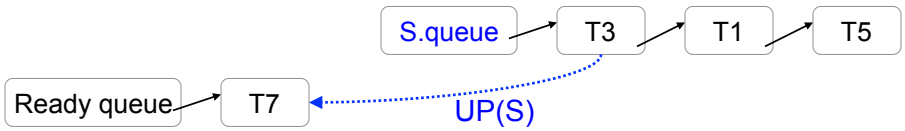
```
DOWN(S):
<disable interrupts>
S.count--;
if (S.count < 0) {
  block this thread
  place this thread in S.queue
}
<enable interrupts>
```

- Threads waiting on a semaphore S:

OS Semaphores – `UP(S)` or `signal(S)`

- The `UP` or signal operation removes one thread from the queue and puts it in the list of ready threads

```
UP(S):
<disable interrupts>
S.count++;
if (S.count <= 0) {
  remove a thread T from S.queue
  place this thread T on Ready list
}
<enable interrupts>
```



OS Semaphores - Observations

- When `S.count >= 0`
  - the number of threads that can execute `wait(s)` without being blocked is equal to `S.count`
- When `S.count < 0`
  - the number of threads waiting on S is equal to `|S.count|`
- Atomicity and mutual exclusion
  - no two threads can be in `wait(S)` and `signal(S)` (on the same `S`) at the same time
  - hence the blocks of code defining `wait(S)` and `signal(S)` are, in fact, critical sections

Using Semaphores to Solve CS Problems

Thread Ti:

```
DOWN(S);
<critical section CS>
UP(S);
<remaining section RS>
```

- To allow only one thread in the CS (mutual exclusion):
  - initialize `S.count` to ____
- To allow *n* threads in the CS, for some *n* > 1:
  - initialize `S.count` to ____

Solving the Earlier Problem

```
/* Thread T1 routine */

void * Count(void *arg)
{
    int i;

    for (i=0; i<10; i++)
    {

        cnt++;

    }
    return NULL;
}
```

```
/* Thread T2 routine */

void * Count(void *arg)
{
    int i;

    for (i=0; i<10; i++)
    {

        cnt++;

    }
    return NULL;
}
```

- Solution: use Semaphores to impose mutual exclusion to executing `cnt++`

How are Races Prevented?

Semaphore mutex; 1

```
/* Thread T1 routine */

void * Count(void *arg)
{
    int i;

    for (i=0; i<10; i++ {
        DOWN(mutex);
        R1 ← cnt
        R1 ← R1+1
        cnt ← R1
        UP(mutex);
    }
    return NULL;
}
```

```
/* Thread T2 routine */

void * Count(void *arg)
{
    int i;

    for (i=0; i<10; i++) {
        DOWN(mutex);
        R2 ← cnt
        R2 ← R2+1
        cnt ← R2
        UP(mutex);
    }
    return NULL;
}
```

Solving the Earlier Problem

Semaphore mutex; 1

```
/* Thread T1 routine */

void * count(void *arg)
{
    int i;

    for (i=0; i<10; i++)
    {
        DOWN(mutex);
        cnt++;
        UP(mutex);
    }
    return NULL;
}
```

```
/* Thread T2 routine */

void * count(void *arg)
{
    int i;

    for (i=0; i<10; i++)
    {
        DOWN(mutex);
        cnt++;
        UP(mutex);
    }
    return NULL;
}
```

Activity – Understanding Semaphores

- What are possible values for g, after the three threads below finish executing?

```
/* global shared variable */
int g = 10;

Semaphore s = 0;
```

Thread A

```
g = g + 1;
UP(s);
g = g * 2;
```

Thread B

```
DOWN(s);
g = g - 2;
UP(s);
```

```
int g = 10;
Semaphore s = 0;
```

Thread A

```
R1 = g
R1 = R1+1
g = R1
UP(s);
R2 = g
R2 = R2*2
g = R2
```

Thread B

```
DOWN(s);
R3 = g
R3 = R2-2
g = R3
UP(s);
```

Using OS Semaphores

- Semaphores have two uses:
  - Mutual exclusion: making sure that only one thread is in a critical section at one time
  - Synchronization: making sure that T1 completes execution before T2 starts?

Synchronizing Threads with Semaphores

- Suppose that we have 2 threads: T1 and T2
- How can we ensure that a statement S1 in T1 executes before statement S2 in T2?

```
Semaphore sync;   0
```

Thread T1

```
S1;

UP(sync);
```

Thread T2

```
DOWN(sync);

S2;
```

Activity

- Consider two concurrent threads T1 and T2. T1 executes statements S1 and S3 and T2 executes statement S2

Thread T1	*Thread T2*
S1	S2
S3	

- Use semaphores to ensure that the statements are ***always*** executed in the order S1, S2, S3

Review: Mutual Exclusion

```
Semaphore mutex;   1

Thread T1

      DOWN(mutex);
      critical section
      UP(mutex);

Thread T2

      DOWN(mutex);
      critical section
      UP(mutex);
```

Review: Synchronization

- T2 cannot begin execution until T1 has finished:

```
Semaphore mutex;   0

Thread T1

      Code for T1
      UP(mutex);

Thread T2

      DOWN(mutex);
      Code for T2
```

Deadlock and Starvation

- *Deadlock* occurs when two or more threads are waiting indefinitely for an event that can be caused by only one of the waiting threads
- Let S and Q be two semaphores initialized to 1

Thread T_1	*Thread T_2*
`wait (S);`	`wait (Q);`
`wait (Q);`	`wait (S);`
`...`	`...`
`signal (S);`	`signal (Q);`
`signal (Q);`	`signal (S);`

- *Starvation* – indefinite blocking
  - a thread may never be removed from the semaphore queue in which it is suspended

Summary

- Threads
  - Share global data
- Problem with Threads
  - Races
- Eliminating Races
  - Mutual Exclusion with Semaphores
- Thread Synchronization
  - Use Semaphores
- POSIX Threads and Semaphores

Hands-on Session

- POSIX semaphores
  - `sem_init`
  - `sem_wait`
  - `sem_post`
  - `sem_getvalue`
  - `sem_destroy`
- Fix the `badcnt.c` program to produce correct result
  - Exercise 2 on class website