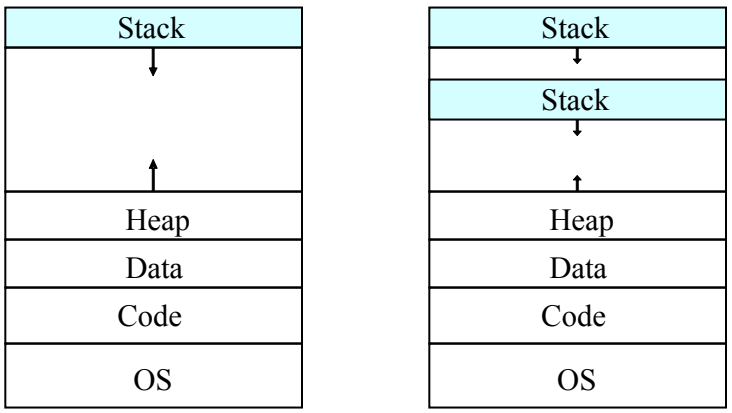
CSC 8400: Computer Systems

Threads and Semaphores

Process Address Space Revisited



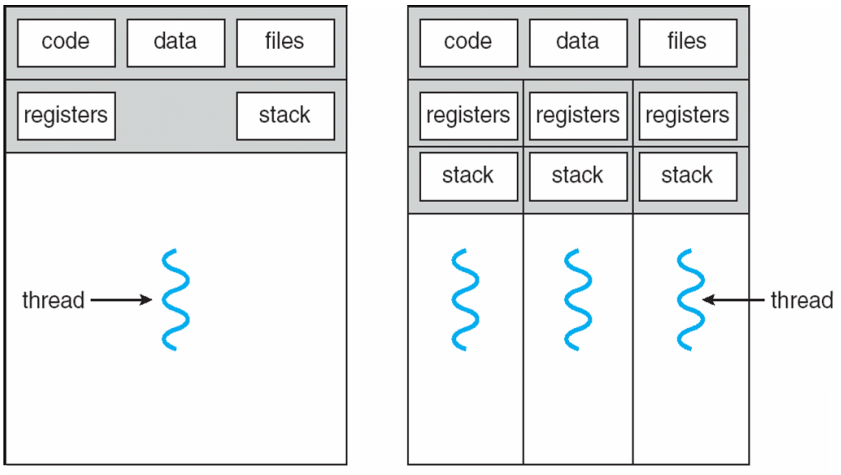
(a) Process with Single Thread

(b) Process with Two Threads

What are Threads?

- A thread is an independent stream of instructions
  - Basic unit of CPU utilization
- A thread has its own
  - thread ID
  - execution stack
- A thread shares with its sibling threads
  - The code, data and heap section
  - Other OS resources, such as open files and signals

Single and Multi-Threaded Processes



single-threaded process multithreaded process

Multi-Threaded Processes

- Each thread has a private stack
- But threads share the process address space!

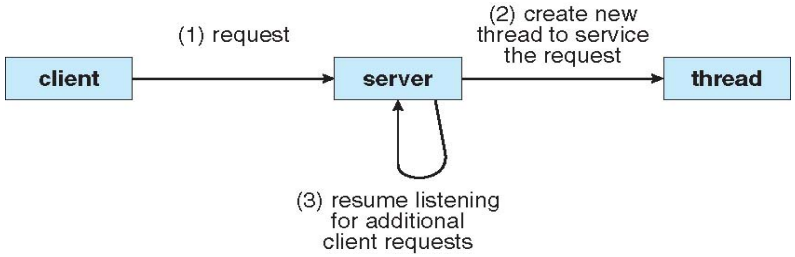
There is no memory protection!

- Threads could potentially write into each other's stack

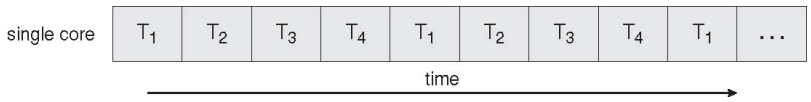
Why use Threads?

- A specific example, a Web server:

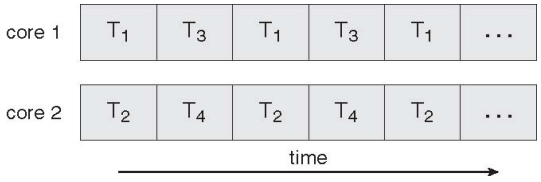
```
in an infinite loop
{       get web page request from client
        check if page exists and client has permissions
        create a thread to transmit web page back to client
}
```



Concurrent Execution on a Single-core System



Parallel Execution on a Multi-core System



POSIX Threads (pthreads)

- Refers to the POSIX standard
- API for thread creation and synchronization
- Common in UNIX operating systems

Java Threads

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- JVM manages Java threads
  - Creation
  - Execution
  - Etc.

Race Conditions
for
Threads

Concurrent Threads

- Concurrent threads may come into conflict with each other when they use shared resources
- Atomic actions are indivisible. In hardware, loads and stores are indivisible.
- On a processor, a thread switch can occur between any two atomic actions; thus the atomic actions of concurrent threads may be interleaved in any possible order.
- Result of concurrent execution should not depend on the order in which atomic instructions are interleaved.

`badcnt.c`: An Incorrect Program

```
#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?!

Critical Sections

- Critical sections are blocks of code that access shared data.

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

- The objective is to make critical sections behave as atomic operations: if one process uses a shared piece of data, other processes can't access it.

Race Condition

- First thread executes cnt++ implemented as

  register1 = cnt
  register1 = register1 + 1
  cnt = register1

- Second thread executes cnt++ implemented as

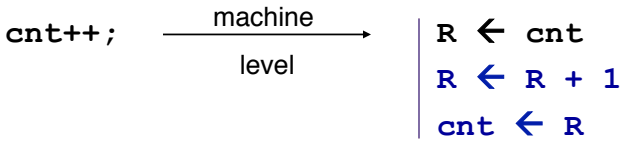
  register2 = cnt
  register2 = register2 + 1
  cnt= register2

- Consider this execution interleaving with "cnt = 5" initially:

  first thread executes register1 = cnt {register1 = 5}
  first thread executes register1 = register1 + 1 {register1 = 6}
  Timer Interrupt
  second thread executes register2 = cnt {register2 = 5}
  second thread executes register2 = register2 + 1 {register2 = 6}
  Timer Interrupt
  first thread executes cnt = register1 {cnt = 6 }
  second thread executes cnt = register2 {cnt = 6}

Race Conditions

- Race Conditions occur when where several threads access shared data concurrently.
- The final value of the shared data may vary from one execution to the next.
- The statement

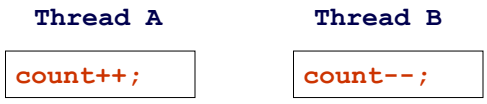


must be performed atomically (cnt is shared)

- An atomic operation is an operation that completes in its entirety, without interruption

Practice Exercise

- Consider two threads sharing a global variable count, initially 10:



- What are the possible values for count after both threads finish executing?

Practice Exercise

- Consider the following three concurrent threads that share a global variable g, initially 10:

Thread A

```
g = g * 2;
```

Thread B

```
g = g - 2;
```

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

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 point x before another thread reaches point y.
- *CS solution*: allow a *single thread* in the CS at one time

Thread T_i

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

```
while(1) {                                   Thread T1
    entry section
        Critical Section (CS)
    exit section
        Remainder Section (RS)
}
```

```
while(1) {                                   Thread T2
    entry section
        Critical Section (CS)
    exit section
        Remainder Section (RS)
}
```

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 called	`wait(S)`
`UP(S)`	also called	`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;
```

- `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 put on the semaphore's queue

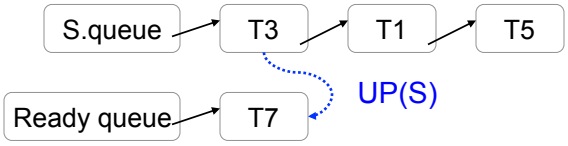
```
DOWN(S):
  <disable interrupts>
  S.count--;
  if (S.count < 0) {
    block this thread
    place this thread in S.queue
  }
  <enable interrupts>
```

- 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 2 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 ____
- What should be the value of s to allow k threads in CS?
  - 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;
}
```

Exercise - Understanding Semaphores

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

```
// global shared variables
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;
```

Exercise

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

T1	T2
S1	S2
S3	

- Use semaphores to ensure that S1 always gets executed before S2 and S2 always gets executed before 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

T_1	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