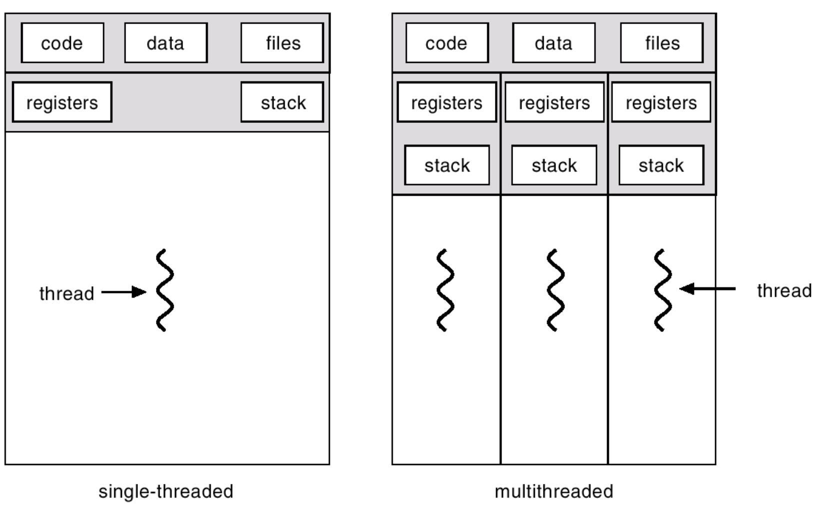
Synchronizing Threads
with
Semaphores

Review: Single vs. Multi-Threaded Processes



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
in = 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!

3

Race conditions

- A *race* occurs when the correctness of the program depends on one thread reaching a statement before another thread.

- No races should occur in a program.

4

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 must have mutual exclusive access to critical sections: two threads cannot simultaneously execute **`cnt++`**

5

The Critical-Section Problem

- To prevent races, concurrent threads must be synchronized. General structure of a thread T_i:

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

- Critical section problem: design mechanisms that allow a single thread to be in its CS at one time

6

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

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

7

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.

8

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>
```

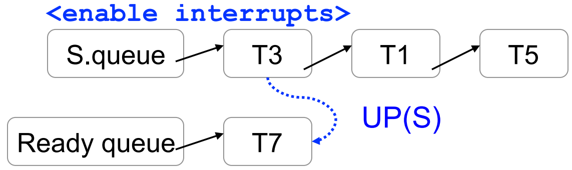
- Threads waiting on a semaphore S:

S.queue → T3 → T1 → T5

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 1
- What should be the value of s to allow k threads in CS?
  - initialize **S.count** to K

13

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;
}
```

- Use Semaphores to impose mutual exclusion to executing cnt++

14

How are Races Prevented?

Semaphore S = 1 ;

```
/* Thread T1 routine */

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

    for (i=0; i<10; i++)
        DOWN (S);          S=0
   1.   R1 ← cnt
   2.   R1 ← R1+1      INT
   3.   cnt ← R1
   4.   UP(S);         S=0
    return NULL;
}
```

```
/* Thread T2 routine */

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

    for (i=0; i<10; i++)
        DOWN (S);          S=-1  BLOCKED !
 5. 3/.  R2 ← cnt
 6.      R2 ← R2+1
 7.      cnt ← R2
 8.      UP (S);
    return NULL;            UNBLOCK
}
```


Exercise - Understanding Semaphores

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

```
// global shared variables
int g = 10;

Semaphore s = 0;
```

Thread A

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

Thread B

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

Thread C

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

~~C~~ A, B

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

Thread A

```
R1 = g
R1 = R1*2
g = R1
```

A.1 { R1 = g; R1 = R1*2 }
A.2 { g = R1 }

Thread B

```
R2 = g
R2 = R2+1
g=R2
UP(s);
```

B.1 { R2 = g }
B.2 { R2 = R2+1; g=R2; UP(s); }

Thread C

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

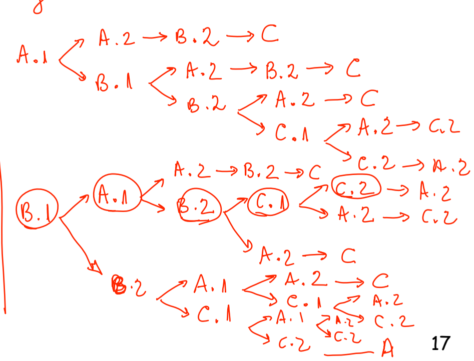
C.1 { R3=g }
C.2 { R3=R2-2; g=R3 }

No interrupts during the execution of a thread:

A, B, C
B, A, C
B, C, A

With interrupts:

A.1, B, C, A.2   g = 20
B, A.1, C, A.2   g = 22
B, C.1, A, C.2   g = 9
B.1, A, B.2, C   g = 9



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?

Using Semaphores to Synchronize Threads

- 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	Thread T2
`S1;`	`DOWN(sync);`
`UP(sync);`	`S2;`

19

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

20

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
```

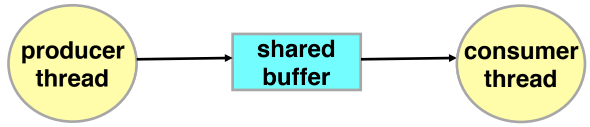
Concurrency Control Problems

- Critical Section (mutual exclusion)
  - only one thread can be in its CS at a time
- Deadlock
  - each thread in a set of threads is holding a resource and waiting to acquire a resource held by another thread
- Starvation
  - a thread is repeatedly denied access to some resource protected by mutual exclusion, even though the resource periodically becomes available

Classical Synchronization Problem

The Producer-Consumer Problem

The Producer – Consumer Problem



- Common synchronization pattern:
  - Producer waits for slot, inserts item in buffer, and "*signals*" consumer.
  - Consumer waits for item, removes it from buffer, and "signals" producer.

- Example: multimedia processing:
  - Producer creates MPEG video frames
  - Consumer renders the frames

Example: Buffer that holds one item (1)

```
/* buf1.c - producer-consumer
on 1-element buffer */

#define NITERS 5

void *producer(void *arg);
void *consumer(void *arg);

typedef struct {
  int buf; /* shared var */
  sem_t full; /* sems */
  sem_t empty;
} sbuf_t;

sbuf_t shared;
```

```
int main() {
  pthread_t tid_producer;
  pthread_t tid_consumer;

  /* initialize the semaphores */
  sem_init(&shared.empty, 0, 1);
  sem_init(&shared.full,  0, 0);

  /* create threads and wait */
  pthread_create(&tid_producer, NULL,
                 producer, NULL);
  pthread_create(&tid_consumer, NULL,
                 consumer, NULL);
  pthread_join(tid_producer, NULL);
  pthread_join(tid_consumer, NULL);

  exit(0);
}
```

Example: Buffer that holds one item (2)

Initially: empty = 1, full = 0.

```
/* producer thread */
void *producer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* produce item */
    item = i;
    printf("produced %d\n",
            item);

    /* write item to buf */
    sem_wait(&shared.empty);
    shared.buf = item;
    sem_post(&shared.full);
  }
  return NULL;
}
```

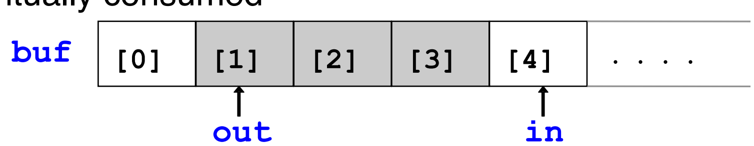
```
/* consumer thread */
void *consumer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* read item from buf */
    sem_wait(&shared.full);
    item = shared.buf;
    sem_post(&shared.empty);

    /* consume item */
    printf("consumed %d\n",
            item);
  }
  return NULL;
}
```


General: Buffer that holds multiple items

- A *circular* buffer **buf** holds items that are produced and eventually consumed




```
#define MAX 10  /* maximum number of slots */

typedef struct {
  int buf[MAX]; /* shared var */
  int in;       /* buf[in%MAX] is the first empty slot */
  int out;      /* buf[out%MAX] is the first busy slot */
  sem_t full;   /* keep track of the number of full spots */
  sem_t empty;  /* keep track of the number of empty spots */
  sem_t mutex;  /* enforce mutual exclusion to shared data */
} sbuf_t;

sbuf_t shared;
```

General: Buffer that holds multiple items

Initially:
empty = MAX,
full = 0.

```
/* producer thread */
void *producer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* produce item */
    item = i;
    printf("produced %d\n",
            item);

    /* write item to buf */
    sem_wait(&shared.empty);
    sem_wait(&shared.mutex);
    shared.buf[shared.in] = item;
    shared.in = (shared.in+1)%MAX;
    sem_post(&shared.mutex);
    sem_post(&shared.full);
  }
  return NULL;
}
```

29

General: Buffer that holds multiple items

```
/* consumer thread */
void *consumer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* read item from buf */
    sem_wait(&shared.full);
    sem_wait(&shared.mutex);
    item = shared.buf[shared.out];
    shared.out = (shared.out+1)%MAX;
    sem_post(&shared.mutex);
    sem_post(&shared.empty);

    /* consume item */
    printf("consumed %d\n", item);
  }
  return NULL;
}
```

30

Summary

- Problem with Threads:
  - Races
- Eliminating Races:
  - Mutual Exclusion with Semaphores
- Thread Synchronization:
  - Use Semaphores
- The Producer-Consumer Problem
- POSIX Threads