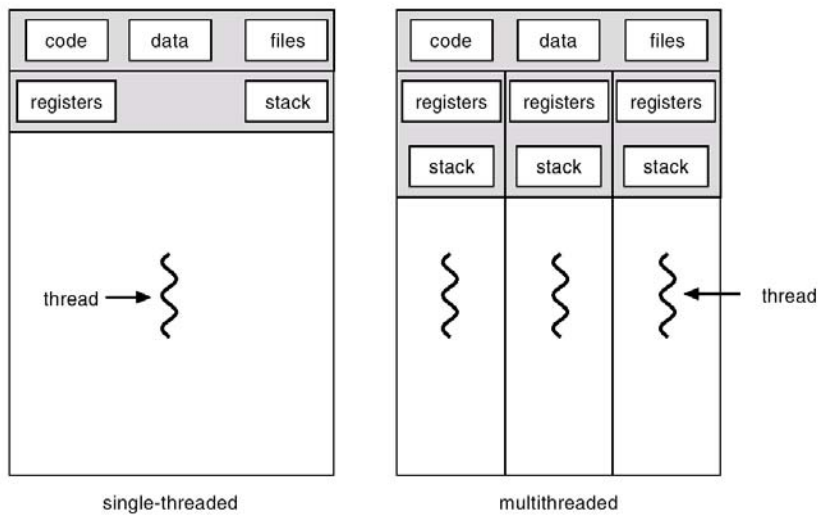


What are Threads?

- Thread
 - Independent stream of instructions
 - Basic unit of CPU utilization
- A thread contains
 - A thread ID
 - A register set (including the Program Counter PC)
 - An execution stack
- A thread shares with its sibling threads
 - The code, data and heap section
 - Other OS resources, such as open files and signals

3

Single and Multi-Threaded Processes



4

Multi-Threaded Processes (2)

- Each thread has a private stack
- But threads share the process address space!
- **There's no memory protection!**
- Threads could potentially write into each other's stack

5

Why use Threads?

- A specific example, a Web server:

```
do
{
    get web page request from client
    check if page exists and client has permissions
    transmit web page back to client
} while(1);
```

- If transmission takes very long time, server is unable to answer other client's requests. Solution:

```
do
{
    get web page request from client
    check if page exists and client has permissions
    create a thread to transmit web page back to client
} while(1);
```

6

Benefits of Threads

- **Responsiveness**
 - Program continues even one part is blocked
- **Resource Sharing**
 - Point to the same process: memory and resources are shared!
- **Economy**
 - Memory allocation for process is costly
 - Context switching for process is costly
- **Utilization of Multiprocessor Architectures**
 - Each thread is assigned one unique processor

7

Threading Issues

- **A badly-behaved thread can damage other threads**
 - Threads share the data of the master process
 - Threads have read/write access to other threads' memory
- **Semantics of fork() and exec() system calls change**
 - Duplicate all threads, or only duplicate only one thread?
 - Implementation-dependent

8

User and Kernel Threads

- **Kernel-Level Threads (KLT, lightweight processes)**
 - Directly supported by the OS
 - Thread is the basic scheduling entity
 - Thread management done by the kernel
 - Examples: Windows 95/98/NT/2000, Solaris, True64 Unix
- **User-Level Threads (ULT)**
 - Implemented as a thread library which contains the code for thread creation, termination, scheduling and switching
 - Kernel is unaware of thread activity
 - Examples: Posix pthreads, Solaris threads

9

pthreads (5.4)

- Refers to the POSIX standard (IEEE 1003.1c)
- API for thread creation and synchronization
- Common in UNIX operating systems

10

Java Threads (5.8)

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

11

Principles of Concurrency

12

Concurrent Threads

- Concurrent threads come into conflict with each other when they come to 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.

13

badcnt.c: An Incorrect Program

```
#define NITERS 100000000
```

```
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=198841183

linux> ./badcnt
BOOM! cnt=198261801

linux> ./badcnt
BOOM! cnt=198269672
```

**cnt should be
equal to 200,000,000.
What went wrong?!**

14

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 if they are atomic operations: if one process uses a shared piece of data, other processes can't access it.

15

The Critical-Section Problem

- A *race* occurs when the correctness of the program depends on one thread reaching point x before another thread reaches point y.
- To prevent races, concurrent processes must be synchronized. General structure of a process P_i :

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

- **Critical section problem:** design mechanisms that allow a single process to be in its critical section at one time

16

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 process has to wait, put it in a *queue of blocked processes* waiting on the semaphore.

17

Semaphores (2)

- In fact, a semaphore is a structure:

```
struct Semaphore {  
    int count;  
    Process * queue;    /* blocked */  
};                    /* processes */  
struct Semaphore S;
```

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

18

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): wait
          sem.t S;
<disable interrupts>
S.count--;
if (S.count < 0) {
    block this process
    place this process in S.queue
}
<enable interrupts>
    
```

- Processes waiting on a semaphore S:



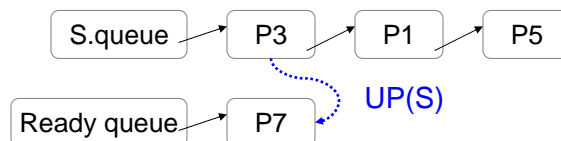
19

OS Semaphores - UP(S) or signal(S)

- The UP or signal operation removes one process from the queue and puts it in the list of ready processes

```

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



20

OS Semaphores - Observations

- When `s.count` ≥ 0
 - the number of processes that can execute `wait(s)` without being blocked is equal to `s.count`
- When `s.count` < 0
 - the number of processes waiting on S is equal to $|\text{s.count}|$
- Atomicity and mutual exclusion
 - no 2 process 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

21

Using Semaphores to Solve CS Problems

Process P_i :

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

- To allow only one process in the CS (mutual exclusion):
 - initialize `s.count` to 1
- What should be the value of `s` to allow `k` processes in the CS?
 - initialize `s.count` to `k`

22

Using OS Semaphores

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

23

Using Semaphores to Synchronize Processes

- Suppose that we have 2 processes: P1 and P2
- How can we ensure that a statement S1 in P1 executes before statement S2 in P2?

Semaphore `sync`; 0

Process P0

```
S1;  
  
UP(sync);
```

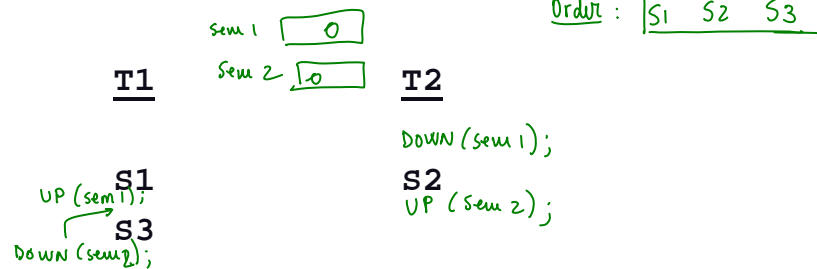
Process P1

```
DOWN(sync);  
  
S2;
```

24

Exercise

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



- Use semaphores to ensure that S1 always gets executed before S2 and S2 always gets executed before S3

25

Review: Mutual Exclusion

```
Semaphore mutex; 1
```

Process P1

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

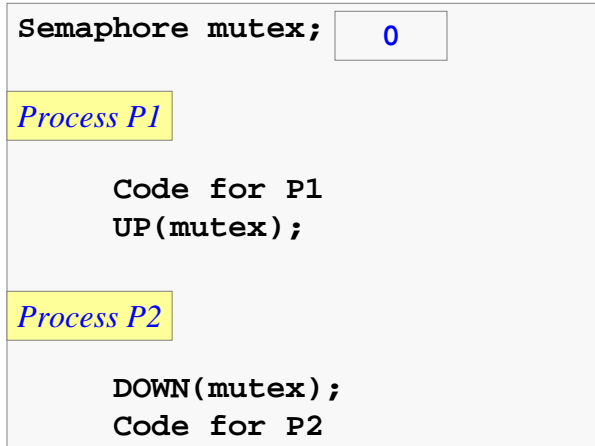
Process P2

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

26

Review: Synchronization

- P2 cannot begin execution until P1 has finished:



27

Concurrency Control Problems

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

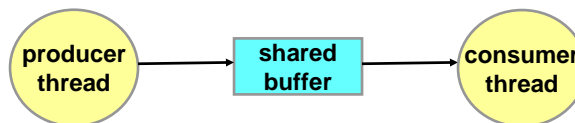
28

Classical Synchronization Problem

The Producer-Consumer Problem

29

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.

■ Examples

- Multimedia processing:
 - Producer creates MPEG video frames, consumer renders the frames
- Event-driven graphical user interfaces
 - Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer.
 - Consumer retrieves events from buffer and paints the display.

30

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);
}
```

31

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;

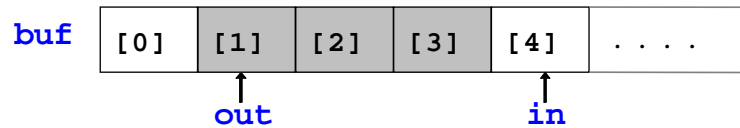
    for (i=0; i<NITERS; i++) {
        /* read item from buf */
        sem_wait(&shared.full);
        item = shared.buf;
        sem_post(&shared.empty);

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

32

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;
```

33

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;
}
```

34

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;
}
```

35

Atomic Operations

- The statement

<code>in++;</code>	→	<code>R = in</code>
	machine	
	level	<code>R = R + 1</code>
		<code>in = R</code>

must be performed atomically (`in` is shared)

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

36

Race Conditions

- One possible interleaving of statements is:

```
R1 = in
R1 = R1 + 1
<timer interrupt ! >
R2 = in
R2 = R2 + 1
in = R2
<timer interrupt !>
in = R1
```

Race condition:

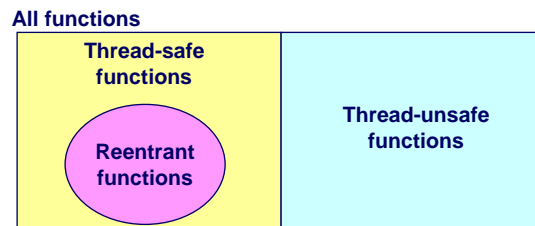
The situation where several processes access shared data **concurrently**. The final value of the shared data may vary from one execution to the next.

- Then `in` ends up incremented once only! Threads overwrite each other's data.

37

Reentrant Functions

- A function is *reentrant* iff it accesses **NO** shared variables when called from multiple threads.
 - Reentrant functions are a proper subset of the set of thread-safe functions.



Tread carefully with threads!

38