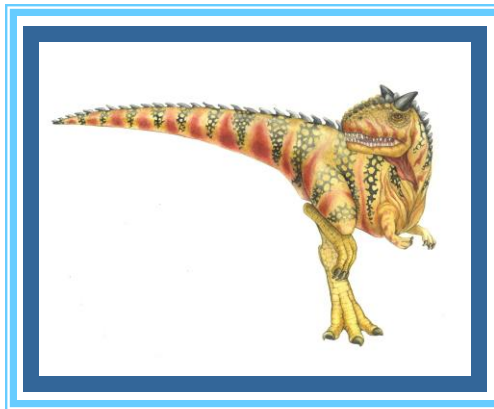


# Chapter 5: Process Synchronization

---





# Discussed till now

---

- n Race condition
  - Demonstration through Producer-Consumer problem
- Critical Section Problem
  - Requirements for solutions to the problem
- Solutions to Critical Section Problem
  - Software solution – Peterson's algorithm
  - Hardware solutions (special atomic instructions) -
    - ▶ test\_and\_set( )
    - ▶ compare\_and\_swap( )





# Mutex Locks





# Mutex Locks

- ❑ Previous hardware-based solutions are complicated and generally inaccessible to application programmers
- ❑ OS designers build **software tools** to solve critical section problem
- ❑ Simplest such tool is mutex lock
  
- ❑ Protect a critical section by first **acquire()** a lock then **release()** the lock
  - ❑ Boolean variable indicating if lock is available or not
  
- ❑ **Calls to `acquire()` and `release()` must be atomic**
  - ❑ Usually implemented via hardware atomic instructions





# acquire() and release()

```
□ acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;;  
}  
  
□ release() {  
    available = true;  
}  
  
□ do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```





# Mutex Locks – adv and disadv

## n Disadvantage

- ❑ Requires **busy waiting** (wastage of CPU cycles that some other process could have used)
- ❑ This type of mutex lock is also called a **spinlock** (process "spins" while waiting for the lock to become available)

## n Advantages

- ❑ No context switch when a process must wait on a lock
- ❑ Useful when locks are expected to be held for short time intervals
- ❑ Can be employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor





# Mutex Locks for threads

---

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                        const pthread_mutexattr_t *attr);
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```





# Mutex Locks for threads (contd.)

---

```
pthread_mutex_t  lock;      // global variable

if (pthread_mutex_init(&lock, NULL) != 0) {
    printf("Error");
    return 1;
}

...

pthread_mutex_lock(&lock);
<critical section>
pthread_mutex_unlock(&lock);

...

pthread_mutex_destroy(&lock);
```







# Semaphores





# Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **s** – integer variable
- Can only be accessed via two indivisible / atomic operations (apart from initialization)
  - **wait()** and **signal()**
  - Originally called **P()** and **V()**
  - P() possibly from "proberen" (to test)
  - V() possibly from "verhogen" (to increase)





# Semaphore operations

- Definition of the wait( ) operation

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

- Definition of the signal( ) operation

```
signal(S) {  
    S++;  
}
```

- Note: both these operations must be **atomic** (executed indivisibly)





# Types of Semaphore

---

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**





# Semaphore Usage

- Can solve various synchronization problems
  - Mutual exclusion
  - Process synchronization
  - Resource counting
- Use a binary semaphore  $s$  (value can be 0 or 1)
  - Initialize  $s$  to 1
  - Every cooperating process has this structure:

$P_i$ :

```
wait(s) ;  
<Critical section>  
signal(s) ;  
<Remainder section>
```





# Semaphore Usage

- Can solve various synchronization problems
  - Mutual exclusion
  - **Process synchronization**
  - Resource counting
  
- Consider  $P_1$  (containing statement  $S_1$ ) and  $P_2$  (containing statement  $S_2$ ) that require  $S_1$  to happen before  $S_2$

Create a semaphore “**synch**” initialized to 0

P1 :

$S_1$ ;

**signal** (**synch**) ;

P2 :

**wait** (**synch**) ;

$S_2$ ;





# Semaphore Usage

---

- Can solve various synchronization problems
  - Mutual exclusion
  - Process synchronization
  - **Resource counting**
  
- We want to keep track of a given resource with a finite number of instances
- Use a counting semaphore whose value can range over an unrestricted domain
  - Initialize to the number of instances available
  - Each process that wishes to use an instance of the resource performs `wait( )`
  - When a process releases its instance of the resource, perform `signal( )`
  
- Can this be used to solve a problem we have discussed earlier?





# Semaphore Implementation

---

- ❑ Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- ❑ Applications may spend lots of time in critical sections; hence important to avoid busy waiting in the implementation of **wait()** and **signal()**







# Avoiding busy waiting

---

- With each semaphore there is an associated waiting queue
- If a process executes **wait(s)** and finds that the semaphore **s** value is not positive, process blocks itself
  - Place the process into the waiting queue for **s**
  - Control transfer to CPU scheduler for selecting next process to execute
- If a process executes **signal(s)**
  - A process that is blocked on **s** should be restarted
  - Move this process from waiting queue to ready queue





# Semaphore Implementation with no busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
- ```
typedef struct{  
    int value;  
    struct process *list;  
} semaphore;
```





# Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

The block() and wakeup() operations must be provided by the kernel as basic system calls





# Some observations

---

- The list of processes waiting for various semaphores can be easily implemented using a link field in the PCB
  - E.g., as a FIFO queue
- The wait( ) and signal( ) operations must execute atomically
  - Disabling interrupts can be a solution for uniprocessor systems (but difficult for multiprocessor systems)
  - Alternative locking via hardware instructions such as TAS or CAS can be used
- Can the value of a counting semaphore become negative?





# Deadlock and Starvation

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

$P_0$   
`wait(S);`  
`wait(Q);`  
`...`  
`signal(S);`  
`signal(Q);`

$P_1$   
`wait(Q);`  
`wait(S);`  
`...`  
`signal(Q);`  
`signal(S);`

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





# Classical Problems of Synchronization

---

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Producer-Consumer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem





# Bounded-Buffer Problem

---

- Producer and consumer processes share the following data:
  - **$n$**  (number of buffers, each of which can hold one item)
  - Semaphore **mutex** initialized to the value 1
  - Semaphore **full** initialized to the value 0
  - Semaphore **empty** initialized to the value  $n$
- Semaphores **empty** and **full** count the number of empty and full buffers
- Semaphore **mutex** provides mutual exclusion in accessing buffer





# Bounded Buffer Problem (Cont.)

- The structure of the **producer process**

```
do {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next produced to the buffer */  
    ...  
    signal(mutex);  
    signal(full);  
} while (true);
```







# Bounded Buffer Problem (Cont.)

- The structure of the **consumer process**

```
do {  
    wait(full);  
    wait(mutex);  
    ...  
    /* remove an item from buffer to next_consumed */  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    /* consume the item in next consumed */  
    ...  
} while (true);
```





# Readers-Writers Problem

---

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
  
- Problem –
  - Multiple readers can be allowed to read at the same time
  - A writer must have exclusive access to the shared data while writing





# Readers-Writers Problem Variations

- Several variations of how readers and writers are considered – all involve some form of priorities
- **First variation** – no reader is kept waiting unless writer has already obtained permission to use shared dataset
  - No reader should wait for other readers to finish, simply because a writer is waiting
  - Readers have priority, Writers may starve
- **Second variation** – once writer is ready, it performs the write ASAP
  - Writer has priority
- Both may have starvation leading to even more variations
- **We discuss a solution to the first readers-writers problem**





# Readers-Writers Problem

---

- Readers and Writers share the following data:
  - Data set
  - Semaphore **rw\_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1
  - Integer **read\_count** initialized to 0
- Semaphore **rw\_mutex** used as mutual exclusion semaphore for writers
- Semaphore **mutex** is used to ensure mutual exclusion when **read\_count** is updated





# Readers-Writers Problem (Cont.)

- The structure of a **writer process**

```
do {  
    wait(rw_mutex);  
    ...  
    /* writing is performed */  
    ...  
    signal(rw_mutex);  
} while (true);
```





# Readers-Writers Problem (Cont.)

- The structure of a **reader process**

```
do {  
    wait(mutex);  
    read_count++;  
    if (read_count == 1)  
        wait(rw_mutex);  
    signal(mutex);  
    ...  
    /* reading is performed */  
    ...  
    wait(mutex);  
    read_count--;  
    if (read_count == 0)  
        signal(rw_mutex);  
    signal(mutex);  
} while (true);
```

If a writer is in the critical section, and  $n$  readers are waiting:

Which readers are queued up on which semaphore?





# Readers-Writers Problem (Cont.)

- The structure of a **reader process**

```
do {  
    wait(mutex);  
    read_count++;  
    if (read_count == 1)  
        wait(rw_mutex);  
    signal(mutex);  
    ...  
    /* reading is performed */  
    ...  
    wait(mutex);  
    read_count--;  
    if (read_count == 0)  
        signal(rw_mutex);  
    signal(mutex);  
} while (true);
```

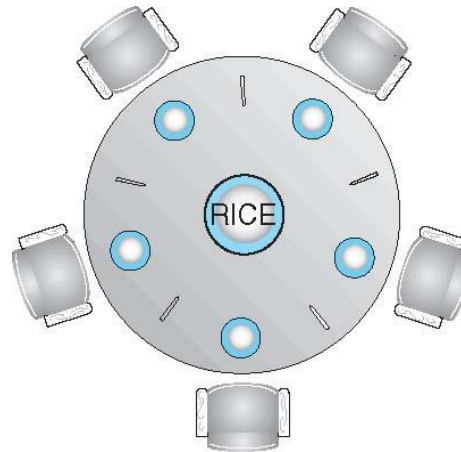
If a writer is in the critical section, and  $n$  readers are waiting:

- one reader is queued on **rw\_mutex**
- the rest  $n-1$  readers are queued on **mutex**





# Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
  - Don't interact with their neighbors
  - When hungry, try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both chopsticks to eat, then release both when done
- In the case of 5 philosophers and 5 single chopsticks
  - Shared data
    - ▶ Semaphore **chopstick** [5] all initialized to 1







# Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {  
    wait (chopstick[i]);  
    wait (chopStick[(i + 1) % 5]);  
  
    /* eat for a while */  
  
    signal (chopstick[i]);  
    signal (chopstick[(i + 1) % 5]);  
  
    /* think for a while */  
  
} while (TRUE);
```

- What is the problem with this algorithm?





# Dining-Philosophers Problem Algorithm (Cont.)

---

- ❑ Can cause deadlock --> all philosophers can starve to death
- ❑ Deadlock handling
  - ❑ Allow at most 4 philosophers to be sitting simultaneously at the table
  - ❑ Allow a philosopher to pick up the chopsticks only if both are available
  - ❑ Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick





# Problems with Semaphores

---

- ❑ Incorrect use of semaphore operations:
  - ❑ signal (mutex) .... wait (mutex)
  - ❑ wait (mutex) ... wait (mutex)
  - ❑ Omitting of wait (mutex) or signal (mutex) (or both)
- ❑ Deadlock and starvation are possible.





# Linux Synchronization

---

- ❑ Linux:
  - ❑ Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - ❑ Version 2.6 and later, fully preemptive
- ❑ Linux provides:
  - ❑ Semaphores
  - ❑ atomic integers
  - ❑ spinlocks
  - ❑ reader-writer versions of both
- ❑ On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption





# POSIX Semaphores

- For using semaphores
  - `#include semaphore.h`
  - Compile the code by linking with `-lpthread -lrt`
- Declare a semaphore: `sem_t mutex;`
- Initialize a semaphore:
  - `sem_init (sem_t *sem, int pshared, unsigned int value)`
  - **sem** - the semaphore to be initialized
  - **pshared** - whether or not the newly initialized semaphore is shared between processes or between threads. Non-zero => the semaphore can be accessed by other processes. Zero => can be shared only by threads belonging to this process
  - **value** - the value to assign to the newly initialized semaphore.
  - E.g., `sem_init( &mutex, 0, 1 );` // initialize to 1





# POSIX Semaphores (contd.)

- Wait on a semaphore
  - `int sem_wait ( sem_t *sem )`
  - E.g., `sem_wait(&mutex);`
  
- Signal a semaphore
  - `int sem_post (sem_t *sem )`
  - E.g., `sem_post(&mutex);`
  
- Destroy a semaphore
  - `sem_destroy(sem_t *sem)`
  - E.g., `sem_destroy(&mutex);`





# Summary

---

- n Race condition
  - Demonstration through Producer-Consumer problem
- Critical Section Problem
  - Requirements for solutions to the problem
- Solutions to Critical Section Problem
  - Software solution – Peterson's algorithm
  - Hardware solutions (special atomic instructions) -
    - ▶ test\_and\_set( )
    - ▶ compare\_and\_swap( )
  - Mutex locks
  - Semaphores
    - ▶ Handling classical problems of synchronization
    - ▶ POSIX semaphores

