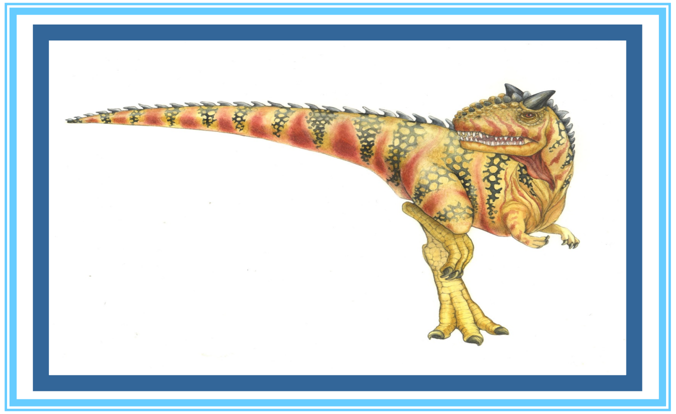
# Spring 2022 COMP 3511
# Review #5



# Coverages

- Semaphore (mutex part will be covered in the POSIX mutex example)
- POSIX Synchronization
- Questions related to synchronization

# Semaphore

- Semaphore ***S*** – non-negative integer variable, can be considered as a generalized lock
  - First defined by Dijkstra in late 1960s. It can behave similarly as mutex lock, but more sophisticated in its usage - the main synchronization primitive used in original UNIX
- Two standard operations modify ***S***: **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**, where **`P()`** stands for "**`proberen`**" (to test) and **`V()`** stands for "**`verhogen`**" (to increment) in Dutch
- It is critical that semaphore operations are executed atomically, which guarantees that no more than one process can execute **`wait()`** and **`signal()`** operations on the same semaphore at the same time.
- The semaphore can only be accessed via these two atomic operations except initialization

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

# Semaphore Usage

- **Counting semaphore** – An integer value can range over an unrestricted domain
  - Counting semaphore can be used to control access to a given resource consisting of a finite number of instances; semaphore value is initialized to the number of resource available
- **Binary semaphore** – integer value can range only between `0` and `1`
  - This can behave similar to mutex locks, can also be used in different ways

- This can also be used to solve various synchronization problems
- Consider ***P₁*** and ***P₂*** that shares a common semaphore `synch`, initialized to `0`; it ensures that ***P1*** process executes ***S₁*** before ***P2*** process executes ***S₂***

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

# 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 on the queue
- 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
- Semaphore values may become negative, whereas this value can never be negative under the classical definition of semaphores with busy waiting.
- If a semaphore value is negative, its magnitude is the number of processes currently waiting on the semaphore.

# Semaphore Implementation with no Busy waiting (Cont.)

```
typedef struct{
    int value;
    struct process *list;
} semaphore;
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);
    }
}
```

Noticing that

- Increment and decrement are done before checking the semaphore value, unlike the busy waiting implementation
- The **`block()`** operation suspends the process that invokes it.
- The **`wakeup(P)`** operation resumes the execution of a suspended process P.

# A Sample Synchronization Question

- Is binary semaphore equivalent as a mutex lock (Yes/No)? Briefly explain your answer:

# Answer

- Is binary semaphore equivalent as a mutex lock (Yes/No)? Briefly explain your answer:

- **Answer**: No (1 mark). (Explanation: 1 mark) Binary semaphore initialized to 1 can be used as a mutex lock, but it can be used for other purposes (e.g., when initialized to 0).

# A Sample Synchronization Question

- Suppose that there are three processes `P0`, `P1` and `P2`

  that need to access a critical section in turn strictly following the order:

  `P0, P1, P2, P0, P1, P2, P0, P1, P2, …`

- In other words, we want to execute P0's critical section, and then P1's critical section, and then P2's critical section, and the pattern repeats

- In the following slides, you will see a program with missing BLANKS

# A Sample Synchronization Question

- Goal: access a critical section in turn strictly following the order: `P0, P1, P2, P0, P1, P2…`

```
// Initialize
// S0, S1, S2 are
// shared

S0 = BLANK1;
S1 = BLANK2;
S2 = BLANK3;
```

```
// Process P0
while(true) {
   wait(S0);
   // critical section
   BLANK4;
}
```

```
// Process P1
while(true){
   BLANK5;
   // critical section
   BLANK6;
}
```

```
// Process P2
while(true) {
  BLANK7;
  // critical section
  signal(S0);
}
```

# A Sample Synchronization Answers

- Goal: access a critical section in turn strictly following the order: `P0, P1, P2, P0, P1, P2…`

```
// Initialize
// S0, S1, S2 are
// shared

S0 = 1;
S1 = 0;
S2 = 0;
```

```
// Process P0
while(true) {
   wait(S0);
   // critical section
   signal(S1);
}
```

```
// Process P1
while(true){
   wait(S1);
   // critical section
   signal(S2);
}
```

```
// Process P2
while(true) {
  wait(S2);
  // critical section
  signal(S0);
}
```

# Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes (details in Chapter 7)
- Let `S` and `Q` be two semaphores initialized to 1

| $P_0$ | $P_1$ |
|---|---|
| `wait(S);` | `wait(Q);` |
| `wait(Q);` | `wait(S);` |
| ... | ... |
| `signal(S);` | `signal(Q);` |
| `signal(Q);` | `signal(S);` |

- Consider if P0 executes wait(S) and P1 wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q), However, P1 is waiting until P0 execute signal(S). Since these signal() operations will never be executed, P0 and P1 are **deadlocked**, extremely difficult to debug
- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue, in which it is suspended. For instance, if we remove processes from the queue associated with a semaphore using LIFO (last-in, first-out) order or based on certain priorities.

# POSIX Synchronization

- POSIX API provides
  - Mutex locks (Note: pthread library supports mutex locks)
  - Semaphores (Note: pthread library don’t have a direct support on semaphores)
  - condition variables (Note: `pthread_cond_wait` and `pthread_cond_signal` are used to provide a waiting based on a condition)
- Widely used on UNIX, Linux, and MacOS

# POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Acquiring and r

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

# Sample Mutex in Pthread (Question)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
unsigned int counter = 0;

// This constant controls
// the number of iteration
#define TOTAL_ITERATIONS 5000

void *add(void *arg) {
  int i;
  pthread_mutex_t* m = BLANK1;
  for (i=0; i<TOTAL_ITERATIONS; i++) {
    BLANK2;
    ++counter;
    BLANK3;
  }
}
```

```
int main() {
  pthread_t tid1, tid2;
  pthread_mutex_t mutex;
  pthread_mutex_init(&mutex, NULL);
  printf("The original counter value is %d\n", counter);
  pthread_create(&tid1, NULL, add, &mutex);
  pthread_create(&tid2, NULL, add, &mutex);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  printf("The final counter value is %d\n", counter);
  return 0;
}
```

What are the missing BLANKs?

What is the purpose of this program?

# Sample Mutex in Pthread (Solution)

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
unsigned int counter = 0;

// This constant controls
// the number of iteration
#define TOTAL_ITERATIONS 5000

void *add(void *arg) {
  int i;
  pthread_mutex_t* m = (pthread_mutex_t*) arg;
  for (i=0; i<TOTAL_ITERATIONS; i++) {
    pthread_mutex_lock(m);
    ++counter;
    pthread_mutex_unlock(m);
  }
}
```

```c
int main() {
  pthread_t tid1, tid2;
  pthread_mutex_t mutex;
  pthread_mutex_init(&mutex, NULL);
  printf("The original counter value is %d\n", counter);
  pthread_create(&tid1, NULL, add, &mutex);
  pthread_create(&tid2, NULL, add, &mutex);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  printf("The final counter value is %d\n", counter);
  return 0;
}
```

The purpose of this program is to protect the shared variable counter Both threads will increase the counter by 5,000 times, so the final counter value is 10,000

# POSIX Condition Variables

- POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

# POSIX Condition Variables

- Thread waiting for the condition `a == b` to become true:

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```