# OS examples - Linux Threads

- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - Flags control behavior

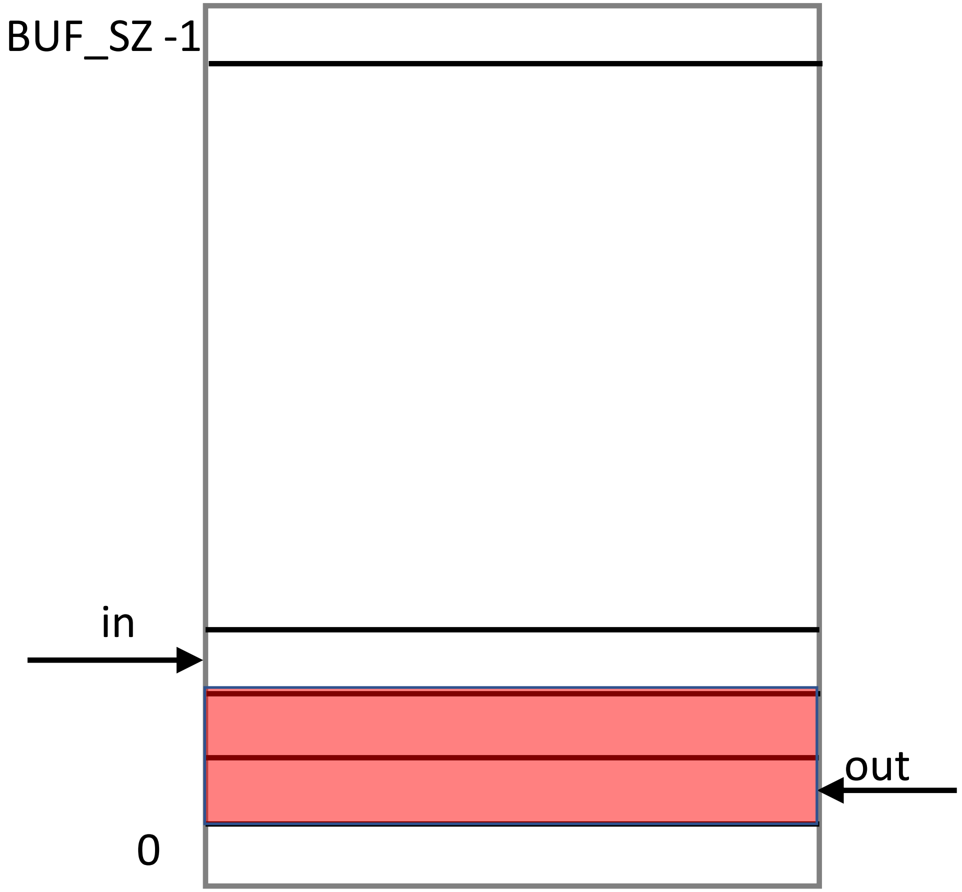
| flag | meaning |
|---|---|
| CLONE_FS | File-system information is shared. |
| CLONE_VM | The same memory space is shared. |
| CLONE_SIGHAND | Signal handlers are shared. |
| CLONE_FILES | The set of open files is shared. |

```
int clone(int (*fn)(void *), void *child_stack,
                 int flags, void *arg, ...
                 /*pid_t *ptid, struct user_desc *tls, pid_t *ctid*/ );
```

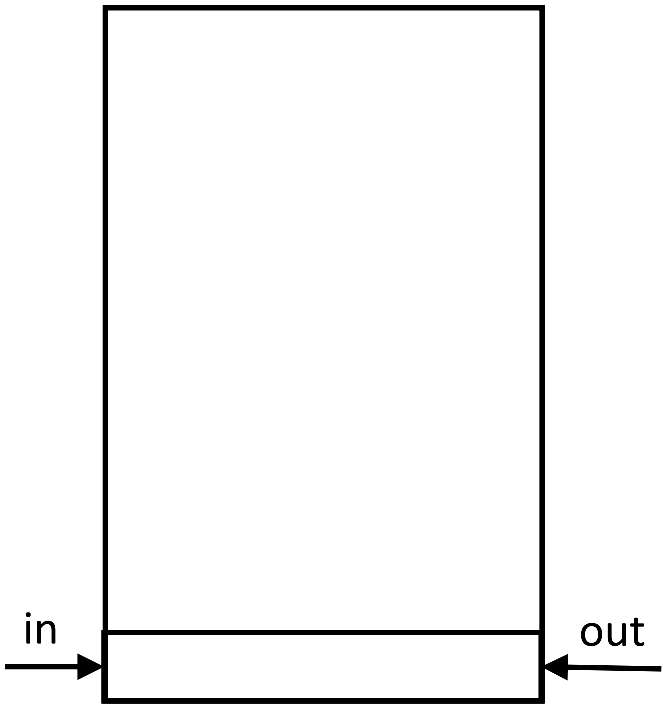
# Consumer-Producer problem – Shared Bounded-Buffer (revisited)

- Shared data:

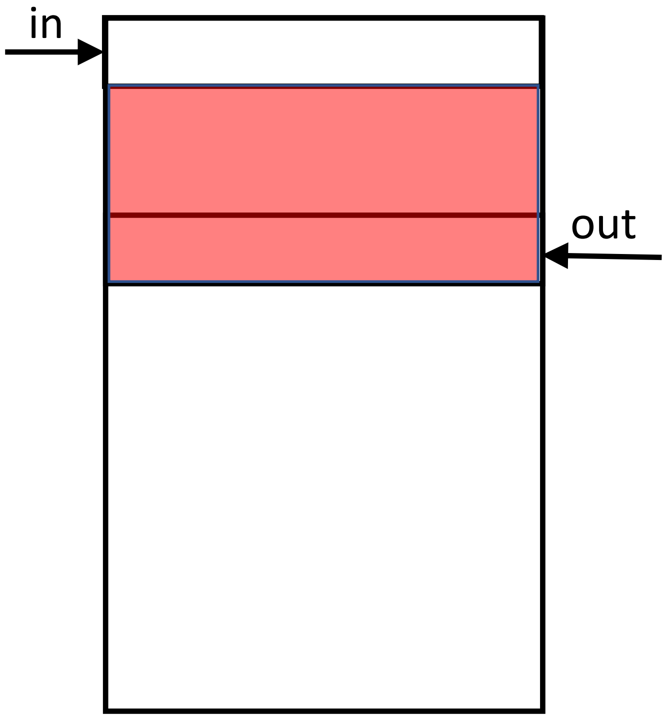
```
#define BUF_SZ 10
typedef struct {
  . . .
} item;

item buffer[BUF_SZ];
int in = 0;
int out = 0;
```



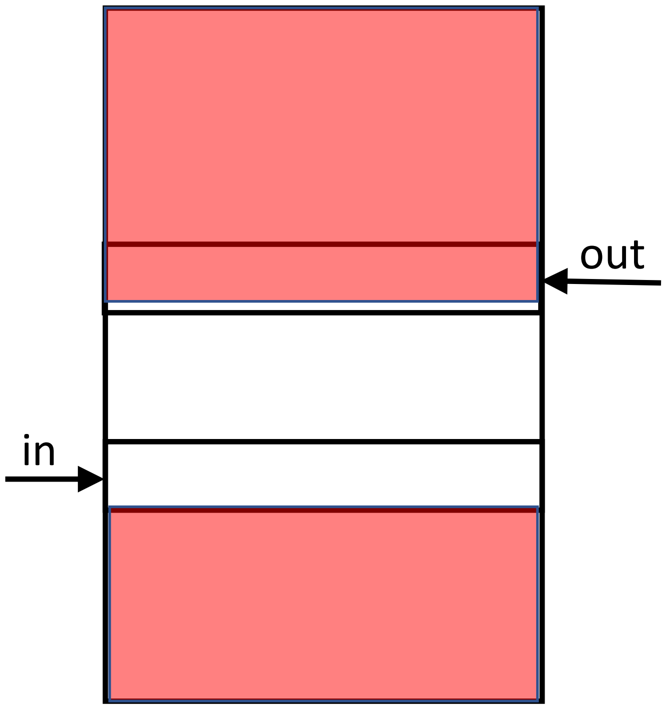

- Buffer needs to be administered and used as a FIFO or Queue




Initial state




After a few writes/reads - No wrapping OR Both ‘in” and “out” wrapped




After a few writes/reads - Only the “in” wrapped but not the “out” index

# Bounded-Buffer – Producer

```
item next_produced;
while (true) {

  /* wait till next in != out */
  while (((in + 1) % BUF_SZ) == out);

  /* produce an item */
  buffer[in] = next_produced;
  in = (in + 1) % BUF_SZ;
}
```

# Bounded Buffer – Consumer

```
item next_consumed;
while (true) {

  /* wait till in != out*/
  while (in == out);

  /* consume an item */
  next_consumed = buffer[out];
  out = (out + 1) % BUF_SZ;

}
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements (why?)
- What happens if both need to access the same location concurrently (e.g. a shared variable or a shared counter?
  - Synchronization  will then be needed

# 5.1 Background

## Producer

```
while (true) {
  /* Wait for room in the shared buffer */
  while (counter == BUFFER_SIZE) ;

  /* Write an entry to shared buffer */
  buffer[in] = next_produced;

  /* Update write pointer and counter */
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}
```

## Consumer

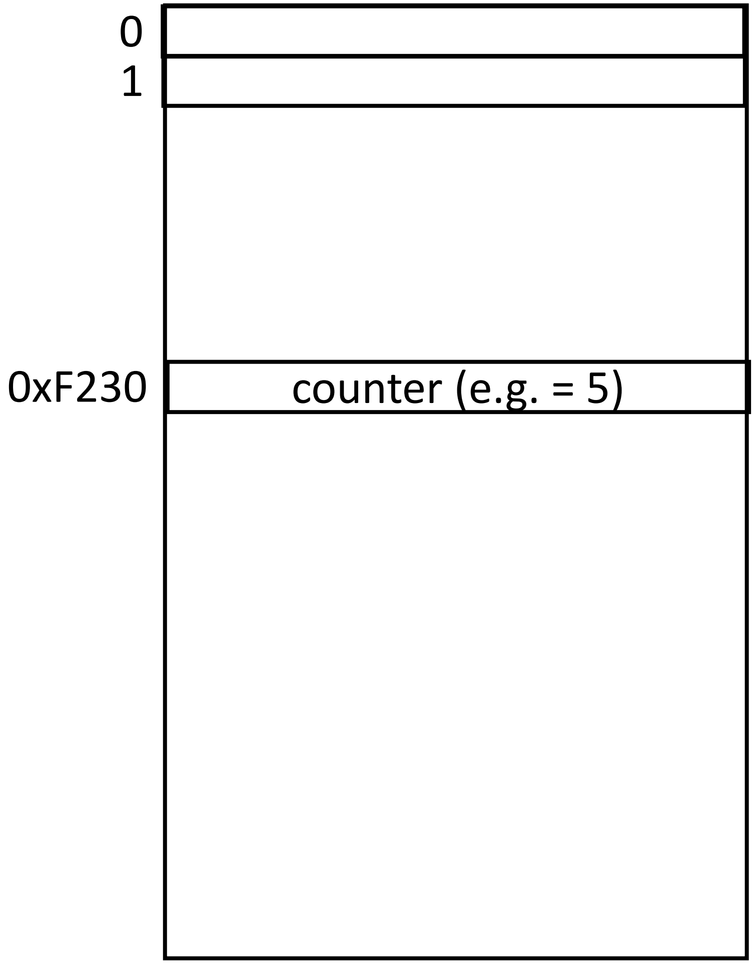
```
while (true) {
  /* Wait for an entry in shared buffer */
  while (counter == 0);

  /* Read an entry from shared buffer */
  next_consumed = buffer[out];

  /* Update read pointer and counter */
  out = (out + 1) % BUFFER_SIZE;
  counter--;
}
```

- Solution uses BUFFER_SIZE elements **but is incorrect** due to the race condition on the shared "`counter`" variable.

# Race Conditions

```
int counter=0;
void thread1()
{
    .
    .

    counter++;

    .
    .
}
```

e.g. if #counter = 0xF230




Memory

```
void thread2()
{
    .
    .

    counter--;

    .
    .
}
```

# Race Conditions

- **`counter++`** could be implemented **in machine code** as

```
   mov r2, #counter

ld  r1,[r2]                    register1 = counter
inc r1                         register1 = register1 + 1
st  r1,[r2]                    counter = register1
```

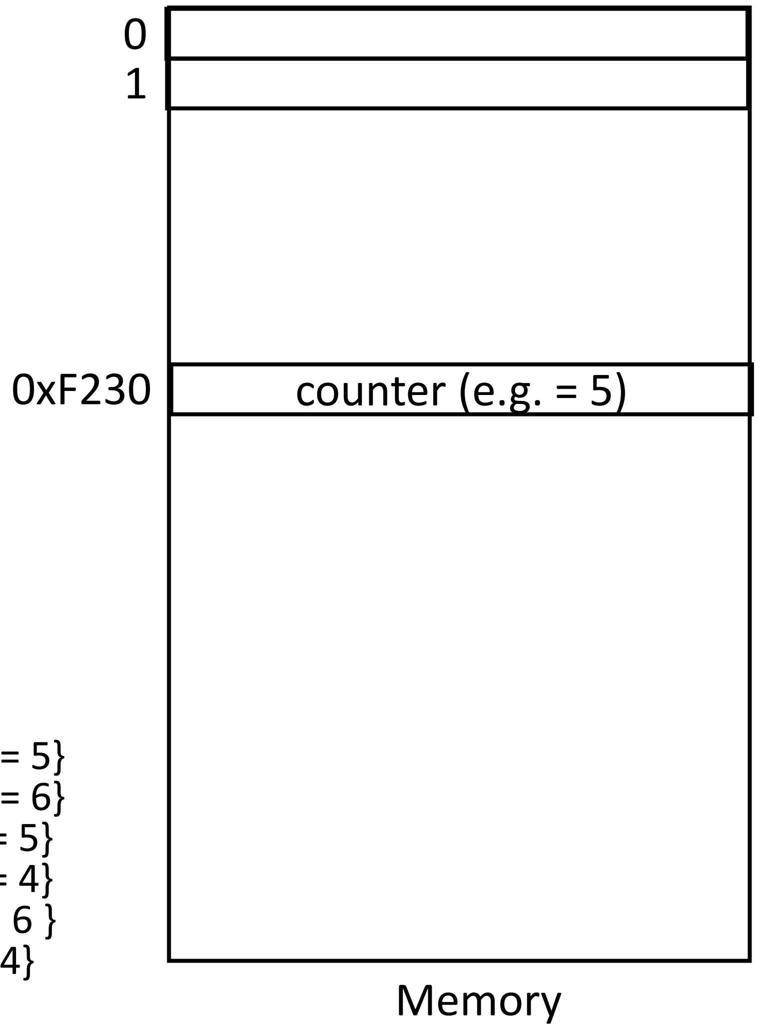
- **`counter--`** could be implemented **in machine code** as

```
mov r3,#counter

ld  r2,[r3]                    register2 = counter
dec r2                         register2 = register2 - 1
st  r2,[r3]                    counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

  S0: producer execute `register1 = counter` {register1 = 5}
  S1: producer execute `register1 = register1 + 1` {register1 = 6}
  S2: consumer execute `register2 = counter` {register2 = 5}
  S3: consumer execute `register2 = register2 - 1` {register2 = 4}
  S4: producer execute `counter = register1` {counter = 6 }
  S5: consumer execute `counter = register2` {counter = 4}

e.g. if #counter = 0xF230



# Race Conditions – cont.

- A **race condition** or **race hazard** is the behavior of a software (or hardware) system where the output is dependent on the sequence or relative timing of the executing threads.
- A **critical race condition** occurs when the order of operations on shared variables causes them to have unexpected or **erroneous** values.
- A **non-critical race condition** occurs when the order of operations on shared variables does not result in an unexpected or erroneous value.
- Critical race conditions result in invalid execution and bugs. Failure to obey mutual exclusion opens up the possibility of corrupting the shared variables.

# Race Conditions - cont.

- A critical race condition occurs when **multiple threads** are performing **non-atomic read-modify-write** concurrently or in parallel.

- It is not necessary for two threads writing to a shared variable concurrently, to result in a critical race condition. A read-modify-write needs to exist to cause a critical race condition.

- Examples of read-modify-write operations:
  - Increment and decrement (e.g. counter++, counter--)
  - test-and-set
  - compare-and-swap
  - accumulate operations (e.g. counter+=4)

# Race Conditions – cont.

- Race conditions have a reputation of being difficult to reproduce and debug, since the end result is nondeterministic and depends on the relative timing between interfering threads.
- Problems occurring in production systems can therefore disappear when running in debug mode, when additional logging is added, or when attaching a debugger. Thus, a bugs that is due to a race condition is often referred to as a "Heisenbug".
- Thus, it is better to avoid race conditions in the first place and there is no alternative to proper and careful software design.

# 5.2 Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code (the section that manipulates shared variables using read-modify-write operations)
  - A Process may be changing common variables, updating a table, writing file, etc
  - To avoid race conditions, when one process is in critical section, no other should be in its critical section
- ***Critical section problem*** requires the design protocol to solve this.
- Each process must ask permission to enter critical section. This happens in the **entry section**. It may follow critical section with an **exit section**, then **remainder section**

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

# Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then:
   - The selection of the processes that will enter the critical section next cannot be postponed indefinitely.
   - Only processes that are in their entry section can participate in the selection.

   --> **no stalls**
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section (and before its request is granted) --> **no starvation**
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

# Critical-Section Handling in OS kernel

- At any instance of time, multiple kernel-mode processes are running concurrently and two or more processes may share data, and thus they have critical sections (where race conditions are possible).
- Two approaches exist for kernel-mode processes:
  - **Preemptive kernels** – allows preemption of process when running in kernel mode (i.e. kernel thread)
    - More responsive since no process can run for too long (thus also suitable for real-time systems)
    - Possibility of race conditions for shared data.
  - **Non-preemptive kernels** – run till kernel mode is exited, and thus blocks (i.e. yields the CPU voluntarily).
    - If kernel code is designed properly, no kernel thread will spend too long in kernel mode.
    - Essentially free of race conditions in kernel mode

# 5.4 Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All H/W solutions described in this section are based on idea of **locking**
  - Protecting critical regions via locks
- **Uniprocessors** – could **disable interrupts**
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems since it requires sending a disable interrupts message to all cores.
    - Operating systems using this approach are not broadly scalable
- Modern machines provide special **atomic hardware instructions**
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

# Solution to Critical-section Problem Using Locks

**Process A**

```
do {
     acquire lock
     critical section
     release lock
     remainder
section
} while (TRUE);
```

**Process B**

```
do {
     acquire lock
     critical section
     release lock
     remainder
section
} while (TRUE);
```

# test_and_set Instruction

Definition:

```
bool test_and_set (bool *target)
{
     bool rv = *target;
     *target = TRUE;
     return rv:
}
```

1. Executed atomically (**it is a single machine instruction**)

**it is a single machine instruction**

1. Returns the original value of the lock variable (`*target`)
2. Set the new value of lock variable (*target) to “TRUE”.

# Using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- A possible solution to critical section problem?

```
do {
   /* Wait till lock is false i.e. not locked, then acquire it */
   while (test_and_set(&lock));

   /* critical section */
   . . .
   /* release the lock at the end (i.e. make it false) */
   lock = false;

   /* remainder section */
   . . .

} while (true);
```

# fetch_and_add Instruction

Definition:

```
int fetch_and_add (int *target, int inc)
{
    int rv = *target;
    *target = *target + inc;
    return rv:
}
```

1. Executed atomically (**it is a single machine instruction**)
2. Returns the original value of the lock variable (*`target`)
3. Set the new value of (*target) to (*target) + inc.

# compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {
    int rv = *value;

    if (*value == expected)
       *value = new_value;
    return rv;
}
```

1. Executed atomically
2. Returns the original value of the lock variable `(*value)`
3. Set the variable “value” the value of the passed parameter “new_value” but only if “*value” ==“expected”. That is, the swap takes place only under this condition.

# Using compare_and_swap

- Shared integer “lock” initialized to 0;
- A possible solution to critical section problem?

```
do {
   /* Wait for value to be zero (i.e. lock is released), then acquire lock */
   while (compare_and_swap(&lock, 0, 1) != 0);

   /* critical section */
   . . .
   /* release the lock when done with CS */
   lock = 0;

   /* remainder section */
   . . .
} while (true);
```

# Bounded-waiting Mutual Exclusion with test_and_set

- **Previous H/W algorithms didn't satisfy the bounded wait requirement.**
- This algorithm uses common data structures:

  `bool waiting[n];`
  `bool lock;`
- The variable `Key` is not shared
- Proof of mutual exclusion:
  - $P_i$ can enter its critical section only if either `waiting[i]== false` OR `key==false`.
  - The value of `key` can become false only if `test_and_set()` is executed. The first process to execute it will find `key == false`; all others must wait.
  - The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;

    /* critical section */
    . . .
    /* Select next process to run
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section starts below*/
    . . .
} while (true);
```

# Bounded-waiting Mutual Exclusion with test_and_set

- Proof of progress:
  - Since a process exiting the critical section either sets `lock` to `false` or sets `waiting[j]` to `false`. Both allow a process that is waiting to enter its critical section to proceed.
- Proof of bounded wait:
  - When a process leaves its critical section, it scans the array waiting in the cyclic ordering ($i + 1$, $i + 2$, ..., $n - 1$, 0, ..., $i - 1$). It designates the first process in this ordering that is in the entry section (`waiting[j] ==true`) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within $n - 1$ turns.

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;

    /* critical section */
    . . .
    /* Select next process to run
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section starts below*/
    . . .
} while (true);
```

**entry section**

**exit section**

# 5.5 Mutex Locks

- The OS provides abstraction for the hardware tools previously described, particularly since they require some shared lock variables.
- Simplest is **mutex**.
- Usage: Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Lock = Boolean variable indicating if lock is available or not

```
int main(){
    do {
        acquire lock
        critical section
        release lock
        remainder section
    } while (true);
}
```

# An implementation using atomic acquire() and release()

- May be implemented via hardware atomic instructions such as
  - `test_and_set` or `compare_and_swap`
- This lock sometimes referred to as a **spinlock** because it requires **busy waiting**, thus
  - **NOT EFFICIENT**.
  - When used, the critical section must be very short
- A mutex may also be implemented without a spinlock by using **wait queues**. The method is explained in the next section for semaphores.

```
int main(){
   do {
      acquire lock
      critical section
      release lock
      remainder section
   } while (true);
}
```

# 5.6 Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- **Theoretically,** it can only be accessed via **indivisible (atomic) operations (shown in blue rectangles)** **`wait()`** and **`signal()`** (Originally called **`P()`** and **`V()`**)

**These are NOT UNIX wait() and signal() API calls**

```
wait(S){
    while(true){   // busy wait till S>0
        if(S > 0){
          S--;
          break;
        }
    }
}
```

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

**S>0 (i.e. 1 and above) indicates that the semaphore is not locked**

**BLUE RECTANGLES indicate atomic operations**

# Semaphore Usage

- **Counting semaphore usage** – integer value can range over an unrestricted domain
  - May be used to organize usage of a resource that only allows access to N processes at a time -> semaphore needs to be initialized to N.
- **Binary semaphore usage**– integer value can range only between 0 and 1
  - Same as a **mutex lock**, except that it has a different polarity **(initialized to 1)**
  - Can synchronization two processes: Consider two processes $P_1$ and $P_2$ that require a statement $S_1$ to happen before $S_2$

    Create a semaphore named "**`synch`**" and **initialize it to 0**

```
P1:
     S1;
     signal(synch);

P2:
     wait(synch);
     S2;
```

- **Note that** generally you cannot initialize the semaphore's value to less than zero. See the man page for `sem_init()`.

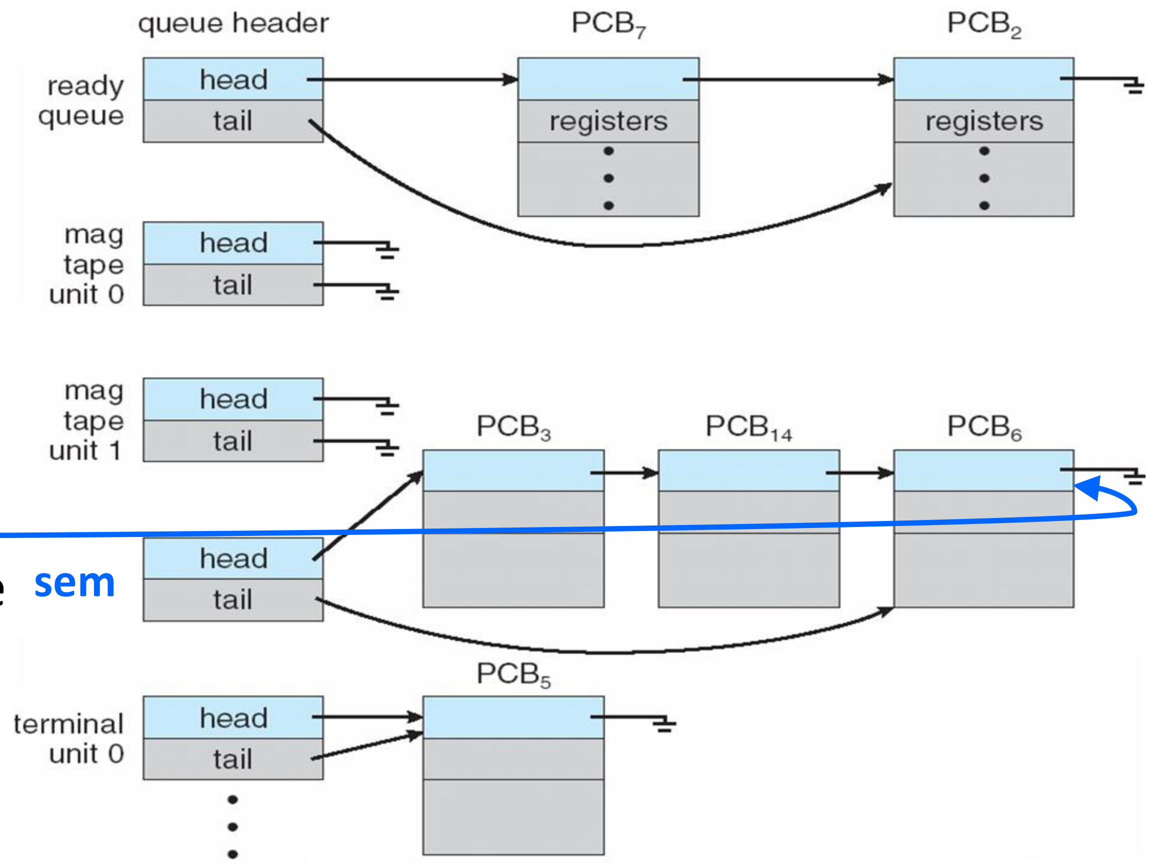
# Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore **concurrently** (i.e. blue rectangles must be guaranteed to execute **atomically**)
- In similarity to a mutex, processes and threads can be **busy waiting** for the semaphore to become available (i.e. >0)
- A process may thus spend a lot of time in entry sections waiting for the semaphore and not doing any useful work.
- Therefore it may be efficient from the system's point of view to block the process and move it into a waiting queue, and schedule another ready process to run instead.

# Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each semaphore has two data items:
  - value (of type integer)
  - pointer to first process in the linked-list queue.
- We define 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();
    }
}
```

Reverse order compared to that used in busy-waiting

**BLUE RECTANGLES indicate atomic operations**

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

# Implementation with no Busy waiting (Cont.)

In this implementation, semaphore values are:

- <0: indicates one or more processes are blocked waiting on the semaphore
  - This is different from previous implementation where the value cannot be <0.
- ==0: indicates the semaphore is not available but no process is blocked waiting on it.
- >0: (i.e. 1 and above) indicates the semaphore is available and thus no process is blocked waiting on it.

# Unix/Pthreads Synchronization

- Named semaphores use names that start with '/' and are less than 251 characters.
  ```
  sem_open
  sem_post
  sem_wait
  sem_close
  sem_unlink
  ```
- Unnamed (anonymous) semaphores use shared variables (for processes or threads) of type `sem_t`.
  ```
  sem_init
  sem_post
  sem_wait
  sem_destroy
  ```

# 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
- 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);` |

# Priority inversion

- **Priority inversion** is a Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Consider having 3 processes L,M and H with low, medium and high priorities respectively, and consider also a resource R that is shared amongst them.
  - If process L acquired R, and then process H requested R, then H will be blocked.
  - If another process M (priority higher than L and is not requesting R) is ready to run, then it may preempt process L (due to the timer tick for example).
  - This indirectly causes priority inversion and it is sometimes problematic.
- Solved via **priority-inheritance protocol**
  - Priority of process L changes to high when H requests the shared resource, and thus won't be preempted by process M.
- This problem occurred on the Mars Pathfinder's robot in 1997 (running a VxWorks real-time OS).