Process Synchronization
Race Condition: Producer-Consumer Problem

while (1) {
    while (count == BUFFER_SIZE)
        ; // do nothing
    // produce an item and put in
    next
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}

Producer Thread

while (1) {
    while (count == 0)
        ; // do nothing
    next = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    // consume the item in next
}

Consumer Thread
Race Condition

- count++ could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2

- Consider this execution: initially, count = 5

Producer:  
register1 = count
register1 = register1 + 1

Consumer:  
register2 = count
register2 = register2 - 1

Values:  
{register1 = 5}
{register1 = 6}
{register2 = 5}
{register2 = 4}
{count = 6}
{count = 4}
Why can such interleaving (race) exist?

- Operating system scheduling is non-deterministic
- Context switch can happen at any arbitrary instructions
- In the previous example, any permutation of interleaving for the producer/consumer is possible
- Correct algorithm should not rely on the order to which threads of execution are scheduled
Example: Shared Linked List

(1) \text{n->next = p->next} \quad \text{(a) m->next = p->next}
(2) \text{p->next = n} \quad \text{(b) p->next = m}

Order: \text{a b 1 2}
Example: Shared Linked List

(1) n->next = p->next
(2) p->next = n

(a) m->next = p->next
(b) p->next = m

Order: a 1 2 b
Critical Section

- What is a critical section?
  - Each thread has a segment of code, in which the thread may be changing common variables, updating shared data structures …
  - Code that must be executed mutually exclusively
  - Contains a race condition

- What is a race condition?
  - Outcome depends on the order threads execute
  - Different outcomes are possible

- Necessary conditions
  - Concurrency
  - Shared data
  - Interference
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.

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 that request is granted.
Mutual Exclusion – Hardware Support

- Interrupt disabling

```c
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```
Mutual Exclusion – Hardware Support

- Special machine instructions
  - Exchange instruction (e.g., xchg)
  - Compare&Exchange instruction

```c
void exchange(int register, int memory)
{
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

Statements are executed atomically

```c
int keyi = 1;
do
    exchange(keyi, lock);
while (keyi!=0)
/* critical section */;
lock = 0;
/* remainder */;
```

spinlock
Mutual Exclusion – Hardware Support

- **Advantages**
  - It is simple and therefore easy to verify
  - Applicable to any number of processes sharing main memory
  - It can be used to support multiple critical sections

- **Disadvantages**
  - Busy-waiting consumes processor time
  - Starvation is possible
  - Deadlock is possible
    - E.g., priority-driven preemptive scheduling
Mutual Exclusion – Software Support

- Semaphore:
  - An integer value used for signalling among processes.
  - Three operations, each of which is *atomic*:
    - initialize
    - decrement
      - P, wait, semWait, or acquire
    - increment
      - V, signal, semSignal, or release
Semaphore Implementation

- Must guarantee that no two processes can execute `acquire()` and `release()` on the same semaphore at the same time
- Thus implementation becomes the critical section problem
  - Busy waiting (spinlock) in critical section implementation
    - Implementation code is short
    - Little busy waiting if critical section rarely occupied
    - More busy waiting → waste of CPU resources
  - Applications may spend lots of time in critical sections
    - Performance issues need to be addressed
Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting 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 Implementation

- Both operations are atomic

```c
acquire(S){
    value--;
    if (value < 0) {
        add this process to list
        block;
    }
}

release(S){
    value++;
    if (value <= 0) {
        remove a process P from
        list
        wakeup(P);
    }
}
```
Semaphore

- **Counting** semaphore
  - An integer value can range over an unrestricted domain

- **Binary** semaphore
  - An integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks

```plaintext
Semaphore S; // initialized to 1
acquire(S);
criticalSection();
release(S);
```
Semaphore

```c
struct semaphore {
    int count;
    queueType queue;
};
void semWait(semaphore s) {
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignal(semaphore s) {
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

sys/ in Xinu: screate.c sdelete.c wait.c signal.c
**Strong/Weak Semaphore**

- A queue is used to hold processes waiting on the semaphore
  - In what order are processes removed from the queue?
- **Strong Semaphores** use FIFO
- **Weak Semaphores** don’t specify the order of removal from the queue
Example of Strong Semaphore

1. Processor A
   - Blocked queue
   - Semaphore: $s = 1$
   - Ready queue: C D B

2. Processor B
   - Blocked queue
   - Semaphore: $s = 0$
   - Ready queue: A C D

3. Processor D
   - Blocked queue: B
   - Semaphore: $s = -1$
   - Ready queue: A C

4. Processor D
   - Blocked queue
   - Semaphore: $s = 0$
   - Ready queue: B A C
Example of Semaphore Mechanism
Problems with Semaphores

- Used in pairs
  - `acquire(mutex) … release(mutex)`
- Incorrect use of semaphore operations:
  - `release (mutex) …. acquire(mutex)`
  - `release (mutex) … release (mutex)`
  - Omitting of `release (mutex) or acquire (mutex)` (or both)
Deadlock and Starvation

- Let $S$ and $Q$ be two semaphores initialized to 1

  
  $P_0$
  
  acquire($S$);
  acquire($Q$);
  ...
  release ($S$);
  release ($Q$);

  $P_1$
  
  acquire ($Q$);
  acquire ($S$);
  ...
  release ($Q$);
  release ($S$);

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

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

- **Priority Inversion** - Scheduling problem when lower-priority process holds a lock needed by higher-priority process
Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
Bounded Buffer

- Buffer has limited slots
- Producer waits if buffer full
- Consumer waits if buffer empty
Bounded-Buffer Problem

- Buffer size is $N$
- Semaphore *mutex*
  - for buffer access control
  - initialized to the value 1
- Semaphore *full*
  - Count full slots
  - initialized to the value 0
- Semaphore *empty*
  - Count empty slots
  - initialized to the value $N$. 
Semaphore Solution for Producer

```java
Item buffer[N];
int in = 0, out = 0;
Semaphore mutex = 1; // buffer access control
Semaphore empty = N; // count empty slots
Semaphore full = 0; // count full slots

public void insert (Item item)
{
    empty.aquire();
    mutex.aquire();

    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % N;

    mutex.release();
    full.release();
}
```
Semaphore Practice for Consumer

Item buffer[N];
int in = 0, out = 0;
Semaphore mutex = 1; // buffer access control
Semaphore empty = N; // count empty slots
Semaphore full = 0; // count full slots

public Item Remove(){

    // semaphore operations?

    // remove an item
    item = buffer[out];
    out = (out + 1) % N;

    // semaphore operations?

    return item;
}

Readers and Writers Problem

- Protect shared data or database
  - A writer has exclusive access to data (no other concurrent writers or readers)
  - Readers can have concurrent access (allow multiple readers)

- Formally
  - \( nw = \# \text{ of active writers} \)
  - \( nr = \# \text{ of active readers} \)
  - Safety condition: \( (nr = 0 \text{ or } nw = 0) \text{ and } nw \leq 1 \)
Solution using Semaphores

```c
int nr = 0;
Semaphore mutex = 1,
    db = 1;

writer() {
    while (1) {
        P(db) // write
        nr++
        if (nr == 1) P(db)
        V(mutex)
        // read
        P(mutex)
        nr--
        if (nr == 0) V(db)
        V(mutex)
    }
}

reader() {
    while (1) {
        P(mutex)
        nr++
        if (nr == 1) P(db)
        V(mutex)
        // read
        P(mutex)
        nr--
        if (nr == 0) V(db)
        V(mutex)
    }
}
```
Solution using Semaphores

```c
int nr = 0;
Semaphore mutex = 1, db = 1;

writer() {
  while (1) {
    P(db) // write
    nr++
    if (nr == 1) P(db)
    V(mutex)
    // read
    P(mutex)
    nr--
    if (nr == 0) V(db)
    V(mutex)
  }
}
```

```c
reader() {
  while (1) {
    P(mutex)
    nr++
    if (nr == 1) P(db)
    V(mutex)
    // read
    P(mutex)
    nr--
    if (nr == 0) V(db)
    V(mutex)
  }
}
```
Solution using Semaphores

- What is the use of:
  - mutex?
  - db?
Dining-Philosophers Problem

- 5 philosophers: Think, or Eat
  - Needs two forks to eat, one from each side
- Shared data
  
  Semaphore fork[] = new Semaphore[5];
Semaphore solution

Semaphore forks[5] =
    {1,1,1,1,1}

philosopher(int i) {
    while (1) {
        fork[i].acquire();
        fork[(i+1)%5].acquire();
        eat;
        fork[i].release();
        fork[(i+1)%5].release();
        think;
    }
}
Semaphore solution

- Problem deadlock
  - How?

- Solution
  - Allow at most 4 sitting
  - Allow one to pick up only if both forks available
  - Break symmetry, e.g., an odd person picks up left first, and even person picks up right first
Priority Inversion

- Real-time priority scheduling
  - Responsiveness: if a higher priority thread appears, serve it asap.

- Mutual exclusion
  - Integrity: if higher priority thread wants to enter critical section being hold by lower priority thread, it has to wait
An Example

- Thread \( \tau_1 \)  L L L \( R_x \) L
- Thread \( \tau_2 \)  L L ... L
- Thread \( \tau_3 \)  L L L \( R_x \) L ... L

- L: local CPU burst
- R: resource required (Mutual Exclusion)
An Example

- Suppose that threads $\tau_1$ and $\tau_3$ share some data
- Access to the data is restricted using semaphore $x$:
  - each task executes the following code:
    - L: do local work
    - P(s): semWait(s)
    - – R: access shared resource
      - V(s): semSignal(s)
    - L: do more local work
An Example

\[ \tau_1 \quad \tau_2 \quad \tau_3 \]

0 \quad t \quad t+3 \quad t+4 \quad t+6

Blocked!
An Example
An Example

$\tau_1$  $\tau_2$  $\tau_3$

$0$  $t$  $t+2$  $t+3$  $t+253$  $t+254$

Blocked!
Unbounded Priority Inversion

In the diagram, we see multiple tasks represented as bars with priority levels. The tasks are labeled as $\tau_1, \tau_2, \tau_3, \ldots, \tau_{2-n}$, and the priorities are indicated by 'L' and 'R' (likely for low and high priority, respectively). The diagram illustrates a situation where a task is blocked due to another task's execution, marked as 'Blocked!'. The time points are marked as $t, t+2, t+3, t+2530, t+2540$, indicating the sequence of events or time steps in the system.
The problem...

- Priority and mutual exclusion at the same time
  - => priority inversion.

- How to resolve it? trade-off?
Priority Inheritance

\[ \tau_1 \]

\[ \tau_2 \]

\[ \tau_3 \]

Blocked!

dynamic \( \pi_3 = \pi_1 \)

Priority Inheritance
Priority Inheritance Protocol

Basic Priority Inheritance

- For each resource (semaphore), a list of blocked threads stored in a priority queue.
- A thread $\tau_i$ uses its assigned priority, unless it is in its critical section and blocks some higher priority threads, in which case, thread $\tau_i$ uses (inherits) the highest dynamic priority of all the threads it blocks.
- Priority inheritance is transitive; that is, if thread $\tau_i$ blocks $\tau_j$ and $\tau_j$ blocks $\tau_k$, then $\tau_i$ can inherit the priority of $\tau_k$. 
The Basic **Priority Inheritance Protocol** has two problems:

- **Deadlock** - two threads need to access a pair of shared resources simultaneously. If the resources, say A and B, are accessed in opposite orders by each thread, then deadlock may occur.

- **Blocking Chain** - the blocking duration is bounded (by at most the sum of critical section times), but that may be substantial.
Problem with Locking

- Pessimistic approach
  - Limits concurrency
- Granularity hard to control
  - Fine-granule locking increases concurrency
  - Drawbacks?
    - Deadlock, overhead
- Convoy effect
  - A thread/process holding a lock is preempted and suspended
  - Other threads waiting on the lock unable to progress
- Fault Tolerance
  - A thread/process holding a lock fails
Synchronization without Locks?

- Transactional memory
  - Optimistic approach
  - Processes go ahead and execute critical sections
  - Conflicts detected and resolved afterwards
    - Rollback – undo and redo

- Sample code for producer generating one item
  ```
  atomic {
    buffer[in] = item;
    in = (in + 1) % N;
  }
  ```
STM vs. Locking

- Software Transactional Memory
  - Less deadlock prone
  - High overhead
  - Better concurrency?
  - Constrained operations within critical sections
    - I/O

- Possible to combine both approaches
  - Runtime evaluation of tradeoff
  - Dynamic selection between TM-mode and Lock-mode
One More Synchronization Exercise

- H2O problem
  - H and O atoms created
    - hready() and oready() calls
    - Each atom one process
  - Processes must wait till there are at least two H atoms and one O atom present
    - Then one of them MUST call makeWater()
    - After which, two instances of hready() and one instance of oready() must return
Solution One

semaphore hPresent = 0;
semaphore waitForWater = 0;

void hready() {
    signal(hPresent);
    wait(waitForWater);
}

void oready() {
    wait(hPresent);
    wait(hPresent);
    makeWater();
    signal(waitForWater);
    signal(waitForWater);
    signal(waitForWater);
}


Solution Two

semaphore numHydrogen = 0;
semaphore pairOfHydrogen = 0, oxygen = 0;

void hready(){
    numHydrogen ++;
    if ((numHydrogen % 2) == 0)  signal(pairOfHydrogen);
    wait(oxygen);
}

void oready() {
    wait(pairOfHydrogen);
    makeWater();
    signal(Oxygen);
    signal(Oxygen);
    signal(Oxygen);
}