

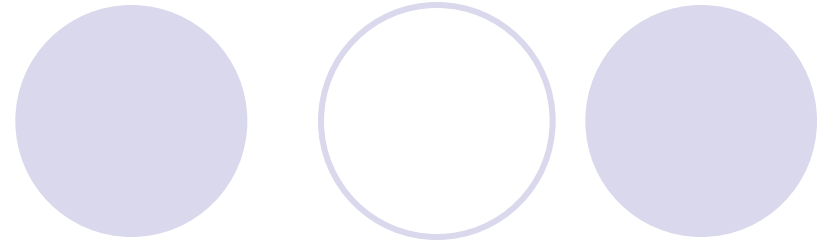
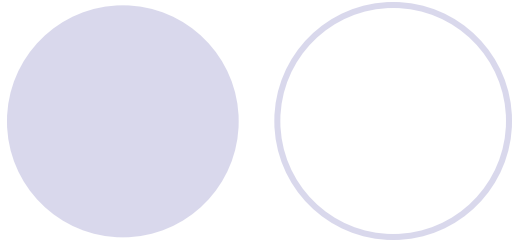
The slide features five light purple circles. Three are solid and two are hollow. They are arranged in two rows: the top row has three circles and the bottom row has two. The text is centered over the top row of circles.

# CE/CZ2005: Operating Systems – Lab Experiment 3



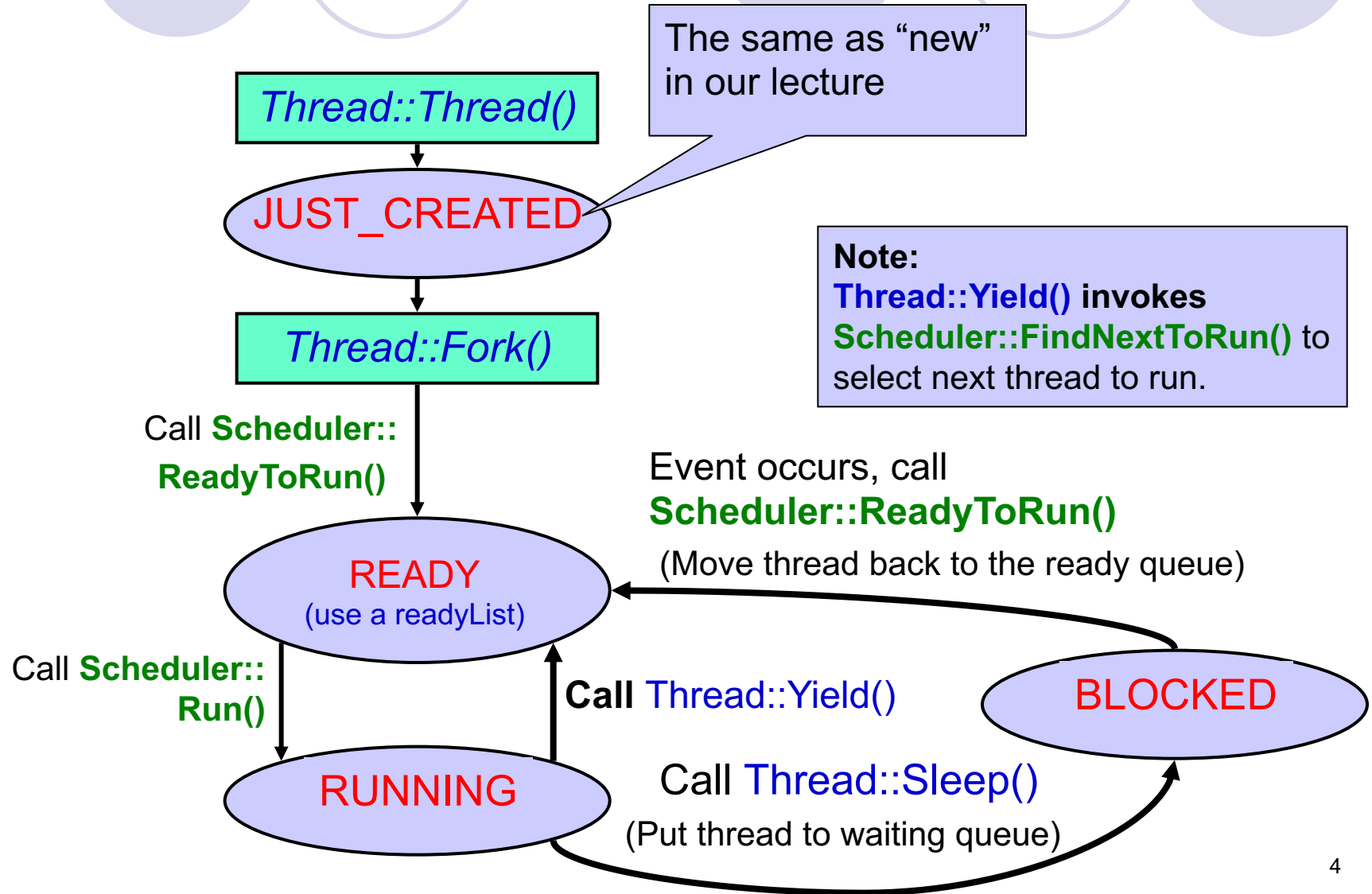
# Outline

- Thread Operations
- Synchronization in NachOS
- Discussion of Experiment 3

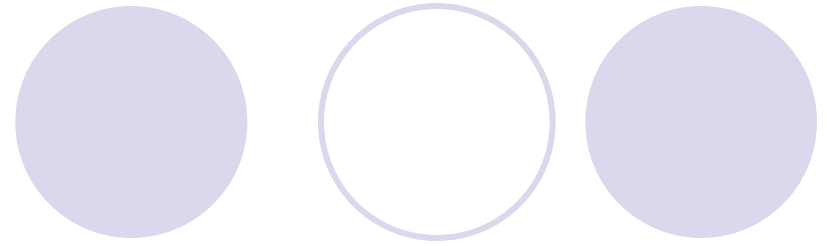


# Thread Operations of NachOS

# Thread Life Cycle



# Thread Object



- **Thread()**
  - Constructor: sets the thread as **JUST\_CREATED** status
- **Fork()**
  - Allocate stack, initialize registers.
  - Call **Scheduler::ReadyToRun()** to put the thread into **readyList**, and set its status as **READY**.
- **Yield()**
  - Suspend the calling thread and put it into **readyList**.
  - Call **Scheduler::FindNextToRun()** to select another thread from **readyList**.
  - Execute selected thread by **Scheduler::Run()**, which sets its status as **RUNNING** and call **SWITCH()** (in **code/threads/switch.s**) to exchange the running thread.
- **Sleep()**
  - Suspend the current thread and find other thread to run
  - Change its state to **BLOCKED**.

# Thread Object (Cont.)

- **Thread \*Thread(char \*debugName)**
  - The *Thread* constructor
    - Setting status to **JUST\_CREATED**,
    - Initializing stack to NULL, and
    - Given the thread name for *debugging*.

# Thread Object (Cont.)

- **Fork(VoidFunctionPtr func, int arg, int joinP);**

- Thread creation

- Allocating stack by invoking *StackAllocate()* function
- Put this thread into ready queue by calling *Scheduler::ReadytoRun()*

- Argument *func*

- The address of a procedure where execution is to begin when the thread starts executing (***The thread's handler function***)

- Argument *arg*

- An integer argument that would be passed to thread handler function

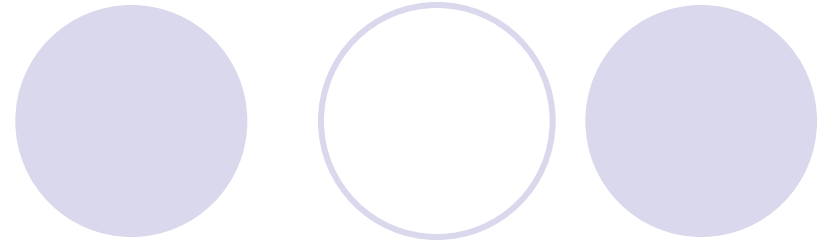
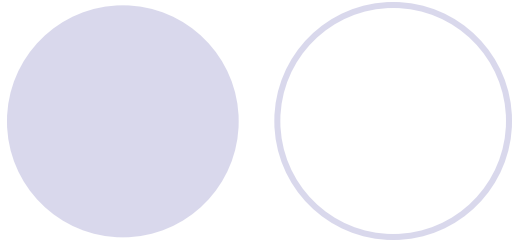
- Argument *joinP*

- Indicate if a Join is going to happen. If joinP=1, a join is going to happen.

# Thread Object (Cont.)

- **void Join (Thread \*forked)**
  - The current thread waits for specified thread to finish before continuing.
  - Argument *forked*
    - Specify the thread to wait for;
    - The thread forked must be a joinable thread (Fork with *joinP=1*).





# Synchronization in NachOS

# Synchronization in NachOS

- There are three synchronization primitives in NachOS:
  1. Semaphores
  2. Locks
  3. Condition variables
- Source code and documentation can be found in
  - `threads/synch.h`
  - `threads/synch.cc`

# Semaphores in NachOS

- A semaphore is a non-negative integer
- Initial value of semaphore depends on number of available resources
- Two operations
  - P() (down/wait) waits until semaphore value  $> 0$  before decrementing it by 1
    - If value is zero, then calling thread is appended to a waiting queue and put to sleep
  - V() (up/signal) increments the semaphore value
    - First thread in waiting queue is put into the ready list

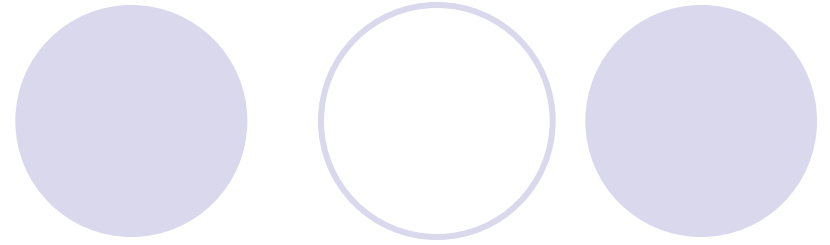
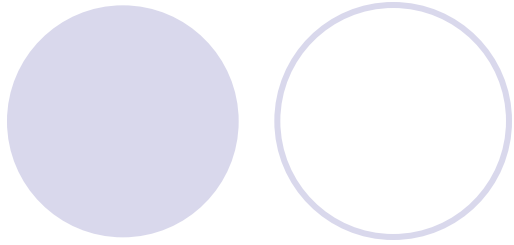
# Locks in NachOS

- Locks are realized by using a binary semaphore
  - A lock can be either FREE or BUSY
- Two operations
  - Acquire()
    - Only one thread can acquire the lock
    - If a lock is busy, other threads have to wait
  - Release()
    - Once a thread releases a lock it will be free again
    - The lock can be acquired by the next thread
- Unlike semaphores, locks are owned by threads
  - Once acquired, a lock has exactly one owner
  - Only the owner can release the lock



# Condition Variables in NachOS

- A condition variable requires a lock
- Three operations
  - Wait(lock)
    - Releases the lock
    - Relinquishes the CPU until signaled
      - Thread is appended to the waiting queue and put to sleep
    - Re-acquires the lock
  - Signal(lock) / notify
    - Wakes up the first thread in the waiting queue (if any)
  - Broadcast(lock) / notifyAll
    - Wakes up all threads in the waiting queue (if any)



## Discussion of Experiment 3

# Experiment 3 – Overview

- Objective

- Understand how to synchronize processes/threads.
- Understand interleavings and race conditions, and master some way of controlling them.
- Know how to use locks/semaphores to solve a critical section problem.

- Tasks

- Implement the race condition scenario for inconsistent output
- Implement the process synchronization scheme for consistent output

# Directory Structure

<b>bin</b>	For generating NachOS format files, <b>DO NOT CHANGE!</b>
<b>filesystem</b>	NachOS kernel related to file system, <b>DO NOT CHANGE!</b>
<b>exp3</b>	<a href="#">Experiment 3, process synchronization.</a>
<b>machine</b>	MIPS H/W simulation, <b>DO NOT CHANGE</b> unless asked.
Makefile.common	For compilation of NachOS,
Makefile.dep	<b>DO NOT CHANGE!</b>
<b>network</b>	NachOS kernel related to network, <b>DO NOT CHANGE!</b>
<b>port</b>	NachOS kernel related to port, <b>DO NOT CHANGE!</b>
readme	Short description of OS labs and assessments
<b>test</b>	NachOS format files for testing virtual memory, <b>DO NOT CHANGE!</b>
<b>threads</b>	NachOS kernel related to thread management, <b>DO NOT CHANGE!</b>
<b>userprog</b>	NachOS kernel related to running user applications, <b>DO NOT CHANGE!</b>
<b>vm</b>	<a href="#"><u>Experiment 4, coding virtual memory (TLB, page replacement)</u></a>



# Experiment 3 – User program

- User program for Experiment 3 can be found in `exp3/threadtest.cc`
  - `ThreadTest()` ← this is the test procedure called from within `main()`
  - You will use it to specify the function to test.

```
//for exercise 1.  
TestValueOne();  
//TestValueMinusOne();  
//for exercise 2.  
//TestConsistency();
```

# Experiment 3 – Task 1

- Arbitrary context switches cause different interleaving execution orders of two threads.
- Without proper process/thread synchronization, a shared variable may have inconsistent value for different interleaving execution orders.
- In this task, we consider a shared variable *value* (initially zero).
- You need to implement the following functions.

```
void Inc_v1(_int which)
void Dec_v1(_int which)
void TestValueOne()
```

After executing TestValueOne, *value=1*

```
void Inc_v2(_int which)
void Dec_v2(_int which)
void TestValueMinusOne()
```

After executing TestValueMinusOne, *value=-1*

Inc\_v1, Inc\_v2, Dec\_v1, Dec\_v2 should follow logic as provided in Inc and Dec. Only allow to add in currentThread -> yield() if needed.

# TestValueOne()

```
//3. implement TestValueOne by create two threads with Inc_v1 and two threads with Dec_v1
// you should pass the checking at the end, printing "congratulations! passed."
void TestValueOne()
{
    value=0;
    printf("enter TestValueOne, value=%d...\n", value);
    //1. fill your code here.
    // An example for thread initialisation is shown.
    Thread *Incl = new Thread("Incl");//, 0, 1);

    //1b. fork all the four joinable threads.
    // An example for thread fork is shown.
    Incl->Fork(Inc_v1, 0, 1);

    //1c. join all the threads (currentThread here refers to main thread)
    // An example is shown where main thread wait for thread Incl to finish running.
    currentThread->Join(Incl);

    //2. checking the value. you should not modify the code or add any code lines behind
    //this section.
    if(value==1)
        printf("congratulations! passed.\n");
    else
        printf("value=%d, failed.\n", value);
}
```

Similar logic and syntax for TestValueMinusOne(), TestConsistency()

# Experiment 3 – Task 2

- With proper process/thread synchronization, shared variable can have consistent value for different interleaving execution orders and different folk orders.
- Continuing Task 1, we consider a shared variable *value* (initially zero).
- You need to implement the following functions, and demonstrate that the consistency is achieved for different interleaving execution orders and different folk orders.

```
void Inc_Consistent (_int which)  
void Dec_Consistent (_int which)  
void TestConsistency ()
```

After executing TestConsistency,  
*value* has a consistent value.

Inc\_Consistent, Dec\_Consistent should follow logic as provided in Inc and Dec. With or without currentThread -> yield(), *value* has a consistent value.

# Synchronization



- Lock
  - Similar syntax as thread for initialisation
- Acquire() and Release() in Inc\_Consistent and Dec\_consistent

# TestConsistency()

```
void TestConsistency()
{
    value=0;
    printf("enter TestConsistency, value=%d...\n", value);

    //fill your code

    //1a. define threads.
    Thread *Inc1 = new Thread("Inc1");//, 0, 1);

    //1b. fork all the four joinable threads.
    // The value should have consistent value regardless of order of thread fork.
    Inc1->Fork(Inc_Consistent, 0, 1);

    //1c. join all the threads
    currentThread->Join(Inc1);

    //2. checking the value. you should not modify the code or add any code lines behind
    //this section.
    if(value==0)
        printf("congratulations! passed.\n");
    else
        printf("value=%d, failed.\n", value);
}
```

# Uncomment all function after done

```
//select the function that you want to test.  
void  
ThreadTest()  
{  
    int loopTimes=0;  
    DEBUG('t', "Entering SimpleTest");  
    //for exercise 1.  
    TestValueOne();  
    TestValueMinusOne();  
    //for exercise 2.  
    TestConsistency();  
}
```

---

# Experiment 3 – Summary

- Objective:

- Understand how to synchronize processes/threads.
- Understand interleavings and race conditions, and master some way of controlling them.
- Know how to use locks/semaphores to solve a critical section problem.

- Assessment:

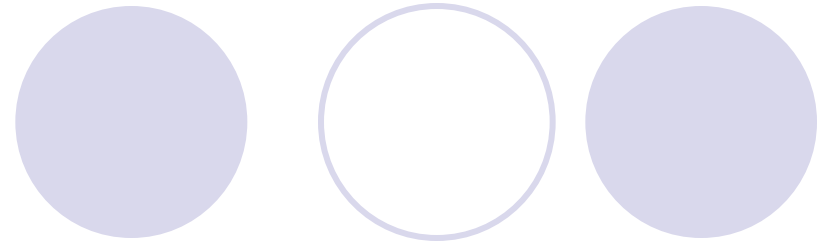
- Assessment of your implementation. Please leave your code in the **exp3** folder for TA/Supervisor to review. **Deadline is 1 week after your lab session (e.g., if lab session is from 10AM-12PM on a Monday, then deadline is 9:59AM on the next Monday).**
- **Lab Quiz 2**, which is an online multiple-choice quiz, will be administered through NTULearn in Week 14.

- Documents:

- Can be found in NTULearn



# Acknowledgement



- The slides are revised from the previous versions created by Dr. Heiko Aydt and Prof He Bingsheng.