Lecture 08
Synchronization Primitives

Dr. Ehtesham Zahoor

The Synchronization Problem

- Processes may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes

- Cooperating process can share data
  - Inter-Process communication in case of heavy-weight processes
  - Same logical address space in case of threads

The Synchronization Problem

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- The problem affects both processes and threads and we will use the term interchangeably.

The Synchronization Problem

- One process/thread should not get into the way of another process/thread when doing critical activities

- Proper sequence of execution should be followed when dependencies are present
    - A produces data, B prints it
    - Before printing B should wait while A is producing data
    - B is dependent on A

If there is no synchronization

Example: Too Much Milk

	Alice	Bob
3:00	Look in fridge - no milk	
3:05	Leave for shop	Look in fridge - no milk
3:10	Arrive at shop	Leave for shop
3:15	Leave shop	Arrive at shop
3:20	Back home - put milk in fridge	Leave shop
3:25		Back home - put milk in fridge
		Oooops!

Problem: Need to ensure that only one process is doing something at a time (e.g., getting milk)

Example: Money Flies Away ...

BALANCE: 2000 €

Bank thread A	Bank thread B
Read a := BALANCE	Read b:= BALANCE
a := a + 500	b := b – 200
Write BALANCE := a	Write BALANCE := b

Oooops!! BALANCE: 1800 €!!!!

Problem: need to ensure that each process is executing its critical section (e.g updating BALANCE) exclusively (one at a time)

ThreadCount: Incorrect implementation

```
#include <pthread.h>
#include <iostream>
#include <unistd.h>
using namespace std;
#define NUM_THREADS 100
int sharedData = 0;
void* incrementData(void* arg)
{
    sharedData++;
    pthread_exit(NULL); }
int main()
{
   pthread_t threadID[NUM_THREADS];
   for (int counter=0; counter<NUM_THREADS;counter++) {
      pthread_create(&threadID[counter], NULL, incrementData, NULL);
   }
   //waiting for all threads
   int statusReturned;
   for (int counter=0; counter<NUM_THREADS;counter++) {
      pthread_join(threadID[counter], NULL);
    }
   cout << "ThreadCount:" << sharedData <<endl;
   pthread_exit(NULL);
}
```

Race condition

- A situation where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place, is called **race condition.**
  - Debugging is not easy
  - Most test runs may run fine

Critical-Section (CS) Problem

- n processes all competing to use some shared data / resource
- Each process has a code segment, called critical section, in which the shared data is accessed

- Problem: ensure that
  - Never two process are allowed to execute in their critical section at the same time
  - Access to the critical section must be an atomic action

Reason behind Race Condition

- Part of the program where the shared memory is accessed is called **Critical Region**
- Races can be avoided
  - **If no two processes are in the critical region at the same time.**

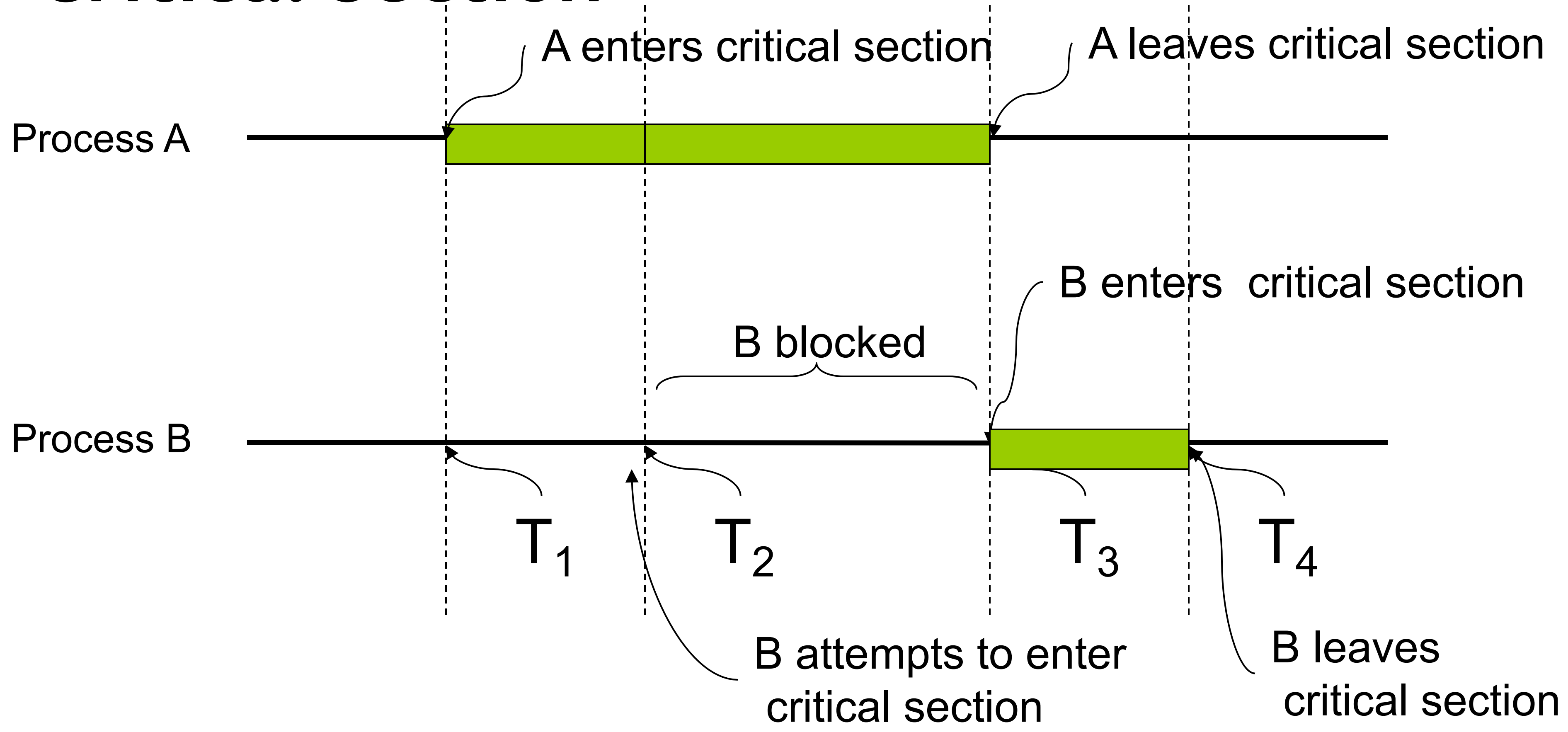
Critical Section

```
void threadRoutine()
{
    int y, x, z;
    y = 3 + 5x;
    y = Global_Var;      }
    y = y + 1;           }
    Global_Var = y;      }
    …                    }
    …
}
```

Critical section?

```
#define NUM_THREADS 100
int sharedData = 0;
void* incrementData(void* arg)
{
    sharedData++;
    pthread_exit(NULL); }
int main()
{
   pthread_t threadID[NUM_THREADS];
   for (int counter=0; counter<NUM_THREADS;counter++) {
      pthread_create(&threadID[counter], NULL, incrementData, NULL);
   }
   //waiting for all threads
   int statusReturned;
   for (int counter=0; counter<NUM_THREADS;counter++) {
      pthread_join(threadID[counter], NULL);
    }
   cout << "ThreadCount:" << sharedData <<endl;
   pthread_exit(NULL);
}
```

Critical Section


A enters critical section
A leaves critical section
Process A
B enters critical section
B blocked
Process B
T_1
T_2
T_3
T_4
B attempts to enter critical section
B leaves critical section

Mutual Exclusion

At any given time, only one process is in the critical section

Critical Section

- Avoid race conditions by not allowing two processes to be in their critical sections at the same time

- We need a mechanism of mutual exclusion

- Some way of ensuring that one processes, whilst using the shared variable, does not allow another process to access that variable

Solution to Critical-Section Problem

- There are 3 requirements that must stand for a correct solution:

1. **Mutual Exclusion**
2. **Progress**
3. **Bounded Waiting**

Solution to CS Problem – Mutual Exclusion

1. **Mutual Exclusion** – If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

- Implications:
  - Critical sections better be focused and short.
  - Better not get into an infinite loop in there.

Solution to CS Problem – Progress

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 process that will enter the critical section next cannot be postponed indefinitely:
   - If only one process wants to enter, it should be able to.
   - If two or more want to enter, one of them should succeed.

Solution to CS Problem – Bounded Waiting

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.
   - Assume that each process executes at a nonzero speed.
   - No assumption concerning relative speed of the *n* processes.

Lock Variables: Software Solution

- Before entering a critical section a process should know if any other is already in the critical section or not
- Consider having a FLAG (also called lock)
- FLAG = FALSE
  - A process is in the critical section
- FLAG = TRUE
  - No process is in the critical section

```
// wait while someone else is in the
// critical region
1. while (FLAG == FALSE);
// stop others from entering critical region
2. FLAG = FALSE;
3. critical_section();
// after critical section let others enter
//the critical region
4. FLAG = TRUE;
5. noncritical_section();
```

Lecture 08: Synchronization Primitives

FLAG = FALSE

Process 1

```
1.while (FLAG == FALSE);
2.FLAG = FALSE;
3.critical_section();
4.FLAG = TRUE;
5.noncritical_section();
```

Process 2

```
1.while (FLAG == FALSE);
2.FLAG = FALSE;
3.critical_section();
```

Timeout

No two processes may be simultaneously inside their critical sections

Process 2 's Program counter is at Line 2

Process 1 forgot that it was Process 2' s turn

Solution: Strict Alternation

- We need to remember “Who’s turn it is?”`
- If its Process 1’s turn then Process 2 should wait
- If its Process 2’s turn then Process 1 should wait

Process 1

```
while(TRUE)
{
  // wait for turn
  while (turn != 1);
  critical_section();
  turn = 2;
  noncritical_section();
}
```

Process 2

```
while(TRUE)
{
  // wait for turn
  while (turn != 2);
  critical_section();
  turn = 1;
  noncritical_section();
}
```

Lecture 08: Synchronization Primitives

Turn = 1

Process 1

```
While(1)
1.while (Turn != 1);

2.critical_section();

3.Turn = 2;
4.noncritical_section();
```

Process 2

```
While(1)
1.while (Turn != 2);

2.critical_section();

3.Turn = 1;
4.noncritical_section();
```

Timeout

Only one Process is in the Critical Section at a time

Process 2 ‘s Program counter is at Line 2

Process 1 Busy Waits

Strict Alternation

Process 1

```
while(TRUE)
{
  // wait
  while (turn != 1);
  critical_section();
  turn = 2;
  noncritical_section();
}
```

Process 1

```
while(TRUE)
{
  // wait
  while (turn != 2);
  critical_section();
  turn = 1;
  noncritical_section();
}
```

- ***Can you see a problem with this?***
- ***Hint : What if one process is a lot faster than the other***

Lecture 08: Synchronization Primitives

turn = 1

Process 1

```
  while(TRUE)
  {
    // wait
    while (turn != 1);
    critical_section();
    turn = 2;
→   noncritical_section();
  }
```

Process 2

```
  while(TRUE)
  {
    // wait
→   while (turn != 2);
    critical_section();
    turn = 1;
    noncritical_section();
  }
```

- Process 1
  - Runs
  - Enters its critical section
  - Exits; setting turn to 2.
- Process 1 is now in its non-critical section.
- Assume this non-critical procedure takes a long time.
- Process 2, which is a much faster process, now runs
- Once it has left its critical section, sets turn to 1.
- Process 2 executes its non-critical section very quickly and returns to the top of the procedure.

Lecture 08: Synchronization Primitives

turn = 1

Process 1

```
while(TRUE)
{
  // wait
  while (turn != 1);
  critical_section();
  turn = 2;
  noncritical_section();
}
```

Process 2

```
while(TRUE)
{
  // wait
  while (turn != 2);
  critical_section();
  turn = 1;
  noncritical_section();
}
```

- Process 1 is in its non-critical section
- Process 2 is waiting for turn to be set to 2
- In fact, there is no reason why process 2 cannot enter its critical region as process 1 is not in its critical region.

Strict Alternation

- What we have is a violation of one of the conditions that we listed above

No process running outside its critical section may block other processes

- This algorithm requires that the processes *strictly alternate* in entering the critical section
- Taking turns is not a good idea if one of the processes is *slower*.

Reason

- Although it was Process 1's **turn**
- But Process 1 was not **interested**.
- Solution:
  - We also need to remember
    - "**Whether it is interested or not?**"

Algorithm 2

- Replace
  - `int turn;`
- With
  - `bool Interested[2];`
- **`Interested[0] = FALSE`**
  - Process 0 is not interested
- **`Interested[0] = TRUE`**
  - Process 0 is interested
- **`Interested[1] = FALSE`**
  - Process 1 is not interested
- **`Interested[1] = TRUE`**
  - Process 1 is interested

Process 0

```
while(TRUE)
{
  interested[0] = TRUE;
  // wait for turn
  while(interested[1]!=FALSE);
  critical_section();
  interested[0] = FALSE;
  noncritical_section();
}
```

Process 1

```
while(TRUE)
{
  interested[1] = TRUE;
  // wait for turn
  while(interested[0]!=FALSE);
  critical_section();
  interested[1] = FALSE;
  noncritical_section();
}
```

Process 0

```
while(TRUE)
{
  interested[0] = TRUE;
  while(interested[1]!=
```

Process 1

```
while(TRUE)
{
      erested[1] = TRUE;
      le(interested[0]!=FALSE);
```

Timeout

DEADLOCK

Peterson's Solution

Combine the previous two algorithms:

```
int turn;
bool interested[2];
```

- **`Interested[0] = FALSE`**
  - Process 0 is not interested
- **`Interested[0] = TRUE`**
  - Process 0 is interested
- **`Interested[1] = FALSE`**
  - Process 1 is not interested
- **`Interested[1] = TRUE`**
  - Process 1 is interested

Peterson’s Solution

Process 0

```
while(TRUE)
{ interested[0] = TRUE;
  turn = 0;
  // wait
  while(interested[ F      &&      T = 0 );
   critical_section();
  interested[0] = FALSE;
  noncritical_section();}
```

Timeout

Process 1

```
while(TRUE)
{
  interested[1] = TRUE;
  turn = 1;
  // wait
  while(interested[ F      &&      T == 1 );

   critical_section();
   interested[1] = FALSE;
   noncritical_section();
   }
```

Peterson's Solution

Process 0

```
while(TRUE)
{ interested[0] = TRUE;
  turn = 0;


    // wait
   while(interested[1 T        && F    0 );

   critical_section();
  interested[0] = FALSE;
  noncritical_section();}
```

Timeout

Process 1

```
while(TRUE)
{
  interested[1] = TRUE;
  turn = 1;
  // wait
  while(interested[ F        && T   = 1 );

   critical_section();
   interested[1] = FALSE;
   noncritical_section();
   }
```

Can not be TRUE at the same time.
Thus used to break tie

Multiple Process Solutions

- Peterson’s solution solves the critical-section problem for two processes, in software
- For multiple processes we have “Bakery Algorithm”
- Used in bakeries etc ??

Bakery Algorithm

- The basic idea is that of a bakery
- On entering the bakery, Customers take tokens
- Whoever has the lowest token gets service next.
- “Service" means entry to the critical section.

Bakery Algorithm

- **`int token[n];`**
- **`token[0]`** = token given to Process 0
- **`token[1]`** = token given to Process 1
- …
- …
- **`token[n-1]`** = token given to Process n – 1

Algorithm

```
while(TRUE)
{
 //1. Receive a token
 token[OwnID]=    max(token[0],token[1],..,token[n-1])+1;
 //2. Wait for turn
 for (OthersID = 0;OthersID<n;OthersID++)
  while(token[OthersID]!=0 &&(token[OthersID],OthersID)<
   (token[OwnID],OwnID));
 //3. Enter Critical section
 critical_section();
 //4. Leave Critical Section
 token[OwnID] = 0;
}
```

Receive a token

- Initially token[0] .. token[n-1] are set to zero
- Process i chooses token[i] as
  - `max(token[0],token[1],...,token[n-1]) + 1;`
- Let n = 5;
- Let the order of execution be P0,P3,P4,P1,P2,P3,P4…
- P0 gets token[0] = max(0,0,0,0,0) + 1 = 0+1=1
- P3 gets token[3] = max(1,0,0,0,0) + 1 = 1+1=2
- P4 gets token[4] = max(1,0,0,2,0) + 1 = 2+1=3
- P1 gets token[1] = max(1,0,0,2,3) + 1 = 3+1=4
- P2 gets token[2] = max(1,4,0,2,3) + 1 = 4+1=5
- P3 gets token[3] = max(1,4,5,2,3) + 1 = 5+1=6
- P4 gets token[4] = max(1,4,5,6,3) + 1 = 6+1=7

Algorithm

Why multiple waits?

```
while(TRUE)
{
 //1. Receive a token
 token[OwnID]=    max(to                              token[n-1])+1;
 //2. Wait for turn
 for (OthersID = 0;OthersID<n;OthersID++)
   while(token[OthersID]!=0 &&(token[OthersID],OthersID)<
   (token[OwnID],OwnID));
 //3. Enter Critical section
 critical_section();
 //4. Leave Critical Section
 token[OwnID] = 0;
}
```

Because, have to wait for multiple processes

Wait for turn

- Pi waits until it has the lowest token of all the processes waiting to enter the critical section.
- Bakery Algorithm does not guarantee that two processes do not receive the same token
- In case of a tie, the process with the lowest ID is served first.

```
for (OthersID = 0;OthersID < n ; OthersID++)
while(token[OthersID]!=0 &&(token[OthersID],OthersID)<
(token[OwnID],OwnID));
```

- (a,b) < (c,d) = TRUE if a < c or if both a = c and b < d
- token[OwnID] = 0 => Process is not trying to enter the critical section

Lecture 08: Synchronization Primitives

```
while(TRUE)
{//1. Receive a token
 token[OwnID]=   max(token[0],token[1],..,token[n-1])+1;
  token[0] = 1
 //2. Wait for turn
  for (OthersID = 0;OthersI T        &&        F
          while(token[OthersID]!=0 &&(token[OthersID],OthersID)<
 (token[OwnID],OwnID));
 //3. Enter Critical section
  critical_section();
```

Timeout

```
while(TRUE)
{//1. Receive a token
 token[OwnID]=   max(token[0],token[1],..,token[n-1])+1;

  token[1] = 1
  //2. Wait for turn
   for (OthersID = 0;Other F        &&        T
           while(token[OthersID]!=0 &&(token[OthersID],OthersID)<
  (token[OwnID],OwnID));
  //3. Enter Critical section
   critical_section();
```

Bakery Algorithm

```
while(TRUE)
{
 //1. Receive a token
 choosing[OwnID] = true;
 token[OwnID]=      max(token[0],token[1],..,token[n-
   1])+1;
 choosing[OwnID] = false;
 //2. Wait for turn
 for (OthersID = 0;OthersID<n;OthersID++)
  while(choosing[OthersID]);
   while(token[OthersID]!=0 &&(token[OthersID],OthersID)<
   (token[OwnID],OwnID));
 //3. Enter Critical section
 critical_section();
 //4. Leave Critical Section
 token[OwnID] = 0;
}
```

Bakery Algorithm

- The reason for `choosing` is to prevent the second **while** loop being *entered* when process $P_{OthersID}$ is setting its `token[OthersID]`.
- `choosing[OthersID]` is true if $P_{OthersID}$ is choosing a token.
- If a process $P_{OthersID}$ is choosing a token when Pi tries to look at it, Pi waits until $P_{OthersID}$ has done so before looking

CS Problem - S/W based solutions

- Bakery and Peterson Solutions solve the critical section problem – in software

- In order to implement an efficient lock why shouldn't we use algorithms such as Peterson or Bakery?
  - Performance considerations
  - Reordering of instructions

FLAG = FALSE

The solution will work, if Testing and Setting the Flag are atomic

Process 1

```
1.while (FLAG == FALSE);
2.FLAG = FALSE;
3.critical_section();
4.FLAG = TRUE;
5.noncritical_section();
```

Process 2

```
1.while (FLAG == FALSE);  }
2.FLAG = FALSE;           }
3.critical_section();
```

Timeout

No two processes may be simultaneously inside their critical sections

Disabling Interrupts

- On a uniprocessor, the only thing that will make multiple instruction sequences not atomic is interrupts.
- So, if want to do a critical section, turn off interrupts before the critical section and turn on interrupts after the critical section.
- Guaranteed atomicity. Early versions of Unix did this.

Hardware Lock

- Some instruction sets assist[illegible]s in implementing mutual exclusio[illegible] multiple processors

```
boolean TestAndSet (boolean *target)
  {
      boolean rv = *target;
      *target = TRUE;
      return rv:
  }
```

Guaranteed to be Atomic

TSL is “Atomic”

- Atomic => Uninterruptible
- No other process can access the memory until TSL is complete
- The CPU executing the TSL instruction locks the memory bus
- Thus, prohibits others CPUs from accessing the memory

Mutual Exclusion Using TSL

- Shared boolean variable lock., initialized to false.
- Solution:

```
do {
        while ( TestAndSet (&lock ))
                ;   // do nothing

                //    critical section

        lock = FALSE;

                //      remainder section

} while (TRUE);
```

…back to threads counting

```
int sharedData = 0;
pthread_mutex_t mutexIncrement;


void* incrementData(void* arg)
{
    pthread_mutex_lock(&mutexIncrement);
    sharedData++;
    pthread_mutex_unlock(&mutexIncrement);


    pthread_exit(NULL); }
int main()
{
  pthread_mutex_init(&mutexIncrement, NULL);


  pthread_t threadID[NUM_THREADS];
   for (int counter=0; counter<NUM_THREADS;counter++) {
      pthread_create(&threadID[counter], NULL, incrementData, NULL);
   }
   //waiting for all threads
   int statusReturned;
   for (int counter=0; counter<NUM_THREADS;counter++) {
      pthread_join(threadID[counter], NULL);
    }
   cout << "ThreadCount:" << sharedData <<endl;
   pthread_exit(NULL);
}
```

Mutex Variables

- A typical sequence in the use of a mutex is as follows:
  - Create and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - Only one succeeds and that thread owns the mutex
  - The owner thread performs some set of actions
  - The owner unlocks the mutex
  - Another thread acquires the mutex and repeats the process
  - Finally the mutex is destroyed, *pthread_mutex_destroy(&mutexvar);*

https://computing.llnl.gov/tutorials/pthreads/#MutexOverview

Disadvantages

- Starvation is possible
  – A process leaves a CS
  – More than one process is waiting

- Deadlock possible if used in priority-based scheduling systems: Ex. scenario
  - Low priority process has the critical region
  - Higher priority process needs it
  - The higher priority process will obtain the CPU to wait for the critical region

Priority Inversion

Consider three processes

- Process L with low priority that requires a resource R
- Process H with high priority that also requires a resource R
- Process M with medium priority

Priority Inversion

- If H starts after L has acquired resource R
  - H has to wait to run until L relinquishes resource R
- Now when M starts
  - M is higher priority unblocked process, it will be scheduled before L
    - Since L has been preempted by M, L cannot relinquish R
- M will run until it is finished, then L will run - at least up to a point where it can relinquish R and then H will run

Priority Inversion

- Thus, in the scenario above, a task with medium priority ran before a task with high priority, effectively giving us a priority inversion.

Possible Solution: Priority inheritance

- If high priority task has to wait for some resource shared with an executing low priority task,
  - the low priority task is temporarily assigned the priority of the highest waiting priority task
  - keeping medium priority tasks from pre-empting the (originally) low priority task, and thereby affecting the waiting high priority task as well.

OpenMP - Synchronization Constructs

- The MASTER directive specifies a region that is to be executed only by the master thread of the team.
- All other threads on the team skip this section of code

```
#pragma omp master newline
…
```

OpenMP - Synchronization Constructs

- The CRITICAL directive specifies a region of code that must be executed by only one thread at a time.
- If a thread is currently executing inside a CRITICAL region and another thread attempts to execute it, it will block until the first thread exits that CRITICAL region.

pragma omp critical [name] newline

…

… back to threadCount

```
int main(int argc, char* argv[])
{
int threadCount;
#pragma omp parallel num_threads(5)
{
  #pragma omp critical
{
int myLocalCount = threadCount;
//sleep(1);
myLocalCount++;
threadCount = myLocalCount;
}
  }
printf("Total Number of Threads: %d\n", threadCount);
}
```

OpenMP - Synchronization Constructs

- When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier.
- All threads then resume executing in parallel the code that follows the barrier.

```
#pragma omp barrier newline
 …
```

Higher-level synchronization primitives

- We have looked at one synchronization primitive: *locks*

- Locks are useful for many things, but sometimes programs have different requirements.

Examples?

- We have a shared variable
  - any number of threads can read the variable
  - but only *one thread* can **write** it at a time.
  - Threads can read after the value has been written

- How it can be done using locks?

Possible Solution?

```
Reader() {
   mycopy = shared_var;
   return mycopy;
}
```

```
Writer() {
   while ( TestAndSet (&lock ));
  shared_var = NEW_VALUE;
  lock = FALSE;
}
```

Another example

- A produces data, B reads it
- Before reading B should wait while A is producing data

- If A and B are sharing a common buffer
- As long as the Buffer is Empty
  - B has to wait
  - Thus, B is dependent on A
- As long as the Buffer is Full
  - A has to wait
  - Thus, A is also dependent on B

Semaphores

- Higher-level synchronization construct
  - Designed by Edsger Dijkstra in the1960's

Semaphore

- Semaphore *S* – integer variable, stored in the kernel, shared by processes
- Two standard operations modify S: wait() and signal()/post()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

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

Atomic

```
signal () / V()
      S++;
   }
```

Atomic

An atomic operation that waits for semaphore to become positive then decrements it by 1

An atomic operation that increments semaphore by 1; wakes up a waiting P, if any

Two uses of Semaphores

- **Mutual exclusion**
  - When semaphores are used for mutual exclusion: “One process in Critical Section”

```
Semaphore mutex;    //  initialized to 1
do {
     P (mutex);
         // Critical Section
     V (mutex);
                                        // remainder section
} while (TRUE);
```

```
wait () / P() {
            while S <= 0; // no-op
            S--;
      }

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

Two uses of Semaphores

- **Scheduling constraints**
  - "When a Process is dependent on another process"
  - Semaphores provide a way for a thread to wait for something.

Binary and Counting Semaphores

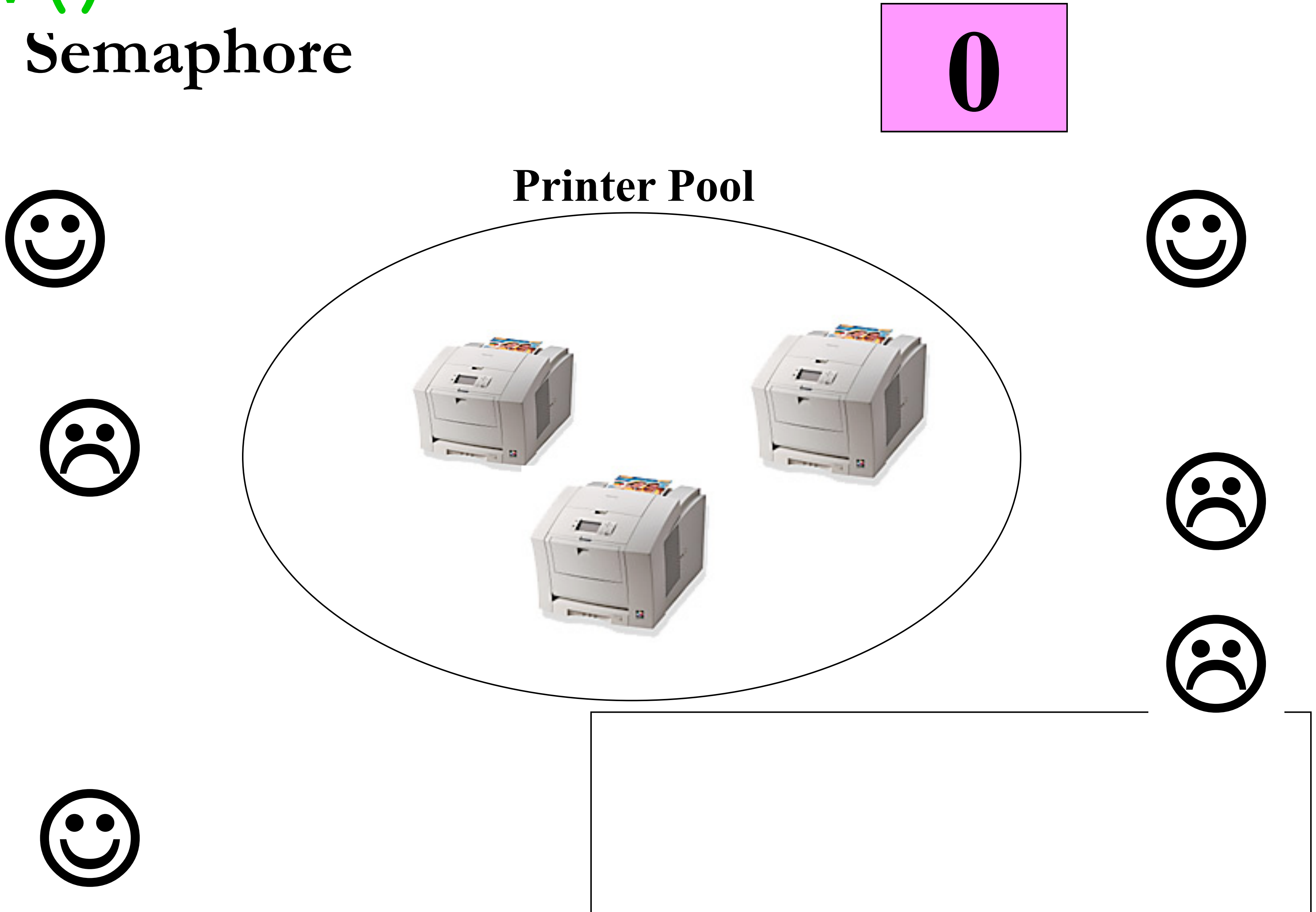
- There are two kinds of Semaphores:
    - Binary semaphores
        - » Control access to a single resource, taking the value of 0 (resource is in use) or 1 (resource is available).
    - Counting semaphores
        - » Control access to multiple resources, thus assuming a range of nonnegative values.

Binary and Counting Semaphores

- The real value of semaphores becomes apparent when the counter can be initialized to a value *other than 0 or 1.*
- Say we initialize a semaphore's counter to 50.
  - What does this mean about P() and V() operations?

V()

Semaphore

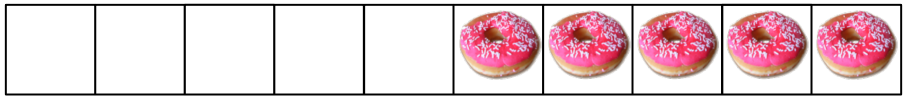


The Producer/Consumer Problem

- Also called the Bounded Buffer problem.

Mmmm… donuts

r



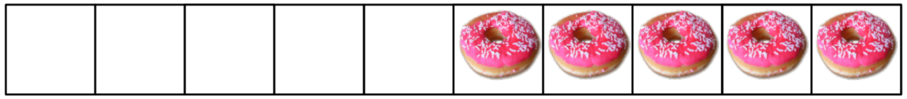
- Producer pushes items into the buffer.
- Consumer pulls items from the buffer.

The Producer/Consumer Problem

- Also called the Bounded Buffer problem.

zzzzz....

r



- Producer pushes items into the buffer.
- Consumer pulls items from the buffer.
- Producer needs to wait when buffer is full.
- Consumer needs to wait when the buffer is empty.

Producer/Consumer Problem

- Consumer must wait for producer to fill buffers, if none full
  - (scheduling constraint)
- Producer must wait for consumer to empty buffers, if all full
  - (scheduling constraint)
- Only one thread can manipulate buffer queue at a time
  - (mutual exclusion)

Use a separate semaphore for each constraint

Producer/Consumer using Sleep/Wakeup

- *Count* keeps track of the number of items in the buffer
- *n* = maximum items in the buffer
- The producer checks against *n*.
  - If *count == n*
    - then the producer sleeps
  - Else
    - Adds the item to the buffer and increments count.
- When the consumer retrieves an item from the buffer
  - If *count == 0*
    - then the consumer sleeps
  - Else
    - Removes an item from the buffer and decrements count.

Producer/Consumer using Sleep/Wakeup

- Calls to WAKEUP occur under the following conditions.
- The producer
  - Adds an item to the buffer
  - Incremented count.
  - if *count = 1* (i.e. the buffer was empty before).
    - wakes up the consumer.
- The consumer
  - Removes an item from the buffer
  - Decrements count.
  - if *count == n-1* (i.e. the buffer was full)
    - Wakes up the producer.

```
int BUFFER_SIZE = 100; int count = 0;
void producer(void) {     int item;
     while(TRUE) {
       produce_item(&item);
       if(count == BUFFER_SIZE)
                   sleep ();
       enter_item(item);
       count++;
       if(count == 1)
                   wakeup(consumer);
     }}
void consumer(void) {     int item;
     while(TRUE) {
       if(count == 0)
                   sleep ();
       remove_item(&item);
       count--;
       if(count == BUFFER_SIZE - 1)
                   wakeup(producer);
       consume_item(&item);
     }}
```

```
int BUFFER_SIZE = 100; int count = 0;
void producer(void) {    int item;
    while(TRUE) {
      produce_item(&item);
      if(count == BUFFER_SIZE)
                  sleep ();
      enter_item(item);
      count++;
      if(count == 1)
                  wakeup(consumer);
    }}
void consumer(void) {    int item;
    while(TRUE) {
      if(count == 0)
                  sleep ();
      remove_item(&item);
      count--;
      if(count == BUFFER_SIZE - 1)
                  wakeup(producer);
      consume_item(&item);
    }}
```

P() is a Generalization of Sleep()

Mutual Exclusion

Scheduling Constraint 2

Scheduling Constraint 1

V() is a Generalization of Wakeup()

```
int BUFFER_SIZE = 100; int count = 0;
void producer(void) {    int item;
    while(TRUE) {
      produce_item(&item);
        Sleep if buffer full
        Get Exclusive Access of Queue
        enter_item(item);
        Leave Exclusive Access of Queue
        Wake up if first item in buffer
    }}
void consumer(void) {    int item;
    while(TRUE) {
        Sleep if buffer empty
        Get Exclusive Access of Queue
        remove_item(&item);
        Leave Exclusive Access of Queue
        Wake up if space in buffer
      consume_item(&item);
```

`P()` is a Generalization of `Sleep()`

Mutual Exclusion

Scheduling Constraint 2

Scheduling Constraint 1

`V()` is a Generalization of `Wakeup()`

Producer/Consumer Problem solved with Semaphores

```
int BUFFER_SIZE = 100; int count = 0;
void producer(void) {    int item;
     while(TRUE) {
       produce_item(&item);
```

Sleep if buffer full

Get Exclusive Access of Queue

```
       enter_item(item);
```

Leave Exclusive Access of Queue

Wake up if first item in buffer

```
     }}
void consumer(void) {    int item;
     while(TRUE) {
```

Sleep if buffer empty

Get Exclusive Access of Queue

```
       remove_item(&item);
```

Leave Exclusive Access of Queue

Wake up if space in buffer

```
       consume_item(&item);
     }}
```

```
Semaphore
Empty(BUFFER_SIZE);
Semaphore full(0);
Semaphore CountMutex(1);
```

Mutual Exclusion

Scheduling Constraint 2

Scheduling Constraint 1