Lecture Notes 5

Process Synchronization

- Producer Consumer – Incorrect implementation of Producer Consumer

```
//Producer Code
while(true){
  while(counter == BUFFER_SIZE);
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}
```

```
//Consumer Code
while(true){
  while(counter == 0);
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter--;
}
```

- Race Condition – The result of multiple processes accessing and manipulating the same data concurrently depends upon order of access
- Process Synchronization and Coordination – The joining up (or handshaking) at specific time to agree upon actions
- Critical Sections – Critical portion of code that only one process can execute concurrently
  - Enter/Exit Section – The code that enters/exits the critical section
  - Mutual Exclusion – If process P_i is executing in critical section no other process is executing in the critical section
  - Progress – Only processes in remainder section can decide on next to enter
  - Bounded Waiting – Limit on the number of times process is "skipped" before entrance to critical section is granted
- Peterson's Solution – Shared memory software solution to critical section problem
  - Two Processes

```
#define FALSE 0
#define TRUE 1
#define N 2
volatile int Turn;
volatile int Interested[N];

void EnterSection(void){
  Interested [Me] = TRUE;
  Turn = Other;
  while(Interested [Other] && Turn == Other);
}

void ExitSection(void){
  Interested [Me] = FALSE;
}
```

- Hardware Support – Special hardware or instructions that support synchronization
  - Lock – A mechanism that prevents other processes from entering locked region
  - Atomic Instructions
    - Test and Set – Atomically tests and sets a memory location

```
EnterSection:
      tsl  lock
      jnz  EnterSection
      ret
```

```
ExitSection:
      move lock, #0
      ret
```

- Swap – Atomically swaps one memory location for register value

```
EnterSection:
      move register, #1
      swap register, lock
      cmp  register, #0
      jne  EnterSection
      ret

ExitSection:
      move lock, #0
      ret
```

- Mutex Locks – Mutual Exclusion Lock
  - Busy Waiting – Constant checking of memory location for change
  - Spinlock – A Mutex Lock implemented using busy wait
- Semaphores – Integer value that is access through atomic operations
  - P, V? (Proberen, Verhogen) or up/down or wait/signal
  - Counting Semaphore – Allows over unrestricted domain
  - Binary Semaphore (Mutex Lock) – Limited to 0 and 1

```
typedef struct{
 int Value;
 struct processqueue Waiting;
} Semaphore;

void down(Semaphore *s){
  s->Value--;
  if(s->Value < 0){
    enqueue(S->Waiting, thisproc);
    block();
  }
}
```

```
void up(Semaphore *s){
  process P;
  s->Value++;
  if(s->Value <= 0){
    P = dequeue(s->Waiting);
    wakeup(P);
  }
}
```

- Deadlock – Processes are waiting upon one another to release resources
- Indefinite Blocking (or Starvation) – Process is waiting to enter critical section, but other processes keep going ahead of it
- Priority Inversion – When a low priority process has a resource and is blocking a high priority process, but cannot release resource because medium priority process is running.
  - Priority-Inheritance Protocol – Protocol to prevent Priority Inversion dynamically
  - Priority Ceiling Protocol – Protocol to prevent Priority Inversion at design time

- Classic Problems of Synchronization
  - Bounded-Buffer – Producer consumer with a bounded buffer

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
void producer(void){
      int item;
      while(TRUE){
            produce_item(&item);
            down(&empty);
            down(&mutex);
            enter_item(item);
            up(&mutex);
            up(&full);
      }
}
```

```
void consumer(void){
      int item;
      while(TRUE){
            down(&full);
            down(&mutex);
            remove_item(&item);
            up(&mutex);
            up(&empty);
            consume_item(item);
      }
}
```

  - Readers Writers – Unlimited readers, but writer needs mutual exclusion

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;

void reader(void){
   while(TRUE){
      down(&mutex);
      rc = rc + 1;
      if(rc == 1) down(&db);
      up(&mutex);
      read_data_base();
      down(&mutex);
      rc = rc - 1;
      if(rc == 0) up(&db);
      up(&mutex);
      use_data_read();
   }
}
```

```
void writer(void){
   while(TRUE){
      think_up_data();
      down(&db);
      write_data_base();
      up(&db);
   }
}
```

- Dining-Philosophers – Each process needs multiple resources
    - N Philosophers, Plates and Forks
    - Need 2 Forks to Eat

```
#define N        5
#define LEFT     (i-1) % N
#define RIGHT    (i+1) % N
#define THINKING 0
#define HUNGRY   1
#define EATING   2

typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i){
   while(TRUE){
      think();
      take_forks(i);
      eat();
      put_forks(i);
   }
}

void take_forks(int i){
   down(&mutex);
   state[i] = HUNGRY;
   test(i);
   up(&mutex);
   down(&s[i]);
}

void put_forks(int i){
   down(&mutex);
   state[i] = THINKING;
   test(LEFT);
   test(RIGHT);
   up(&mutex);
}

void test(int i){
   if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
   EATING){
      state[i] = EATING;
      up(&s[i]);
   }
}
```

- Sleeping Barber – Keep process working when “clients”, sleep when none
    - Single Barber
    - Single Barber Chair
    - N Chairs for Customers

```
#define CHAIRS   5

typedef int semaphore;

semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;
int waiting = 0;

void barber(void){
      while(TRUE){
            down(&customers);
            down(&mutex);
            waiting = waiting -
            1;
            up(&barbers);
            up(&mutex);
            cut_hair();
      }
}

void customer(void){
      down(&mutex);
      if(waiting < CHAIRS){
            waiting = waiting +
            1;
            up(&customers);
            up(&mutex);
            down(&barbers);
            get_haircut();
      }
      else{
            up(&mutex);
      }
}
```

- Monitors – Abstract Data Type that provides mutual exclusion inside the monitor
  - Semaphore Implementation
  - Bounded Buffer

```
monitor ProducerConsumer
   condition full, empty;
   integer count;
   procedure enter;
   begin
      if count = N then
      wait(full);
      enter_item;
      count := count + 1;
      if count = 1 then
      signal(empty)
   end;

   procedure remove;
   begin
      if count = 0 then
      wait(empty);
      remove_item;
      count := count - 1;
      if count = N - 1 then
      signal(full)
   end;
   count := 0;
end monitor;
```

```
procedure producer;
begin
   while true do
   begin
      produce_item;
      ProducerConsumer.enter
   end
end;
```

```
procedure consumer;
begin
   while true do
   begin
      ProducerConsumer.remove;
      consume_item
   end
end;
```

- Deadlock – Processes are waiting upon one another to release resources
  - Necessary Conditions for Deadlock
    - Mutual Exclusion – Resource must be held in non-sharable mode
    - Hold and Wait – Process holds a resource and is waiting on another
    - No Preemption – Resources must be voluntarily released
    - Circular Wait – Must have cycle of processes waiting
  - Resource Allocation Graph – Directed graph of acquired and requested resources and processes
    - Request Edge – Directed edge from process to resource
    - Assignment Edge – Directed edge from resource to process
    - Deadlock occurs if cycle in Resource Allocation Graph
  - Deadlock Prevention – Makes sure that all necessary conditions cannot hold
  - Deadlock Avoidance – Requires additional information about all resources a process will require