Operating Systems (UE18CS302)
Unit 2

Aronya Baksy

August 2020

1 Threads

- Threads are fundamental units of CPU execution, with each thread containing its own program counter, register set, stack and an identifier known as the **thread ID**.
- Processes contain single or multiple threads of execution. Multiple threads allow a process to perform multiple tasks at the same time (eg: most modern desktop applications like Web Browsers, Word Processors, spreadsheet programs etc. are multithreaded).
- Process creation is heavy and requires large overheads (mostly from context switching) while thread creation is lightweight with minimal overheads.

1.1 Benefits of Multithreaded Programming

- **Responsiveness**: As threads are independent of each other, one can run even when another one is blocked (ie. waiting). This is useful in GUI applications where each thread can service one part of the UI and keep response times low.
- **Resource Sharing**: Processes share resources using complicated IPC mechanisms, but threads can share resources easily as they use the memory address space of the process that created them.
- **Economy**: Thread creation has minimal memory allocation and context switching overheads compared to process creation as threads share a common memory, while processes need their own memory and context.
- **Scalability**: A multi-threaded processes can run on multiple processing cores, while a single-threaded process can run only on one core (irrespective of how many cores are there in hardware).

1.2 Multicore Programming (Concurrency vs Parallelism)

- Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency
- If an application has 4 threads, then on a single CPU system, the 4 threads will be interleaved but at a single instant, only one thread will be executed.
- On a multicore system, a separate core can be assigned to run each thread, hence the threads can run in parallel.
- **Concurrent** systems support the execution of more than one task, but only one task is executed at a single instant of time.
- **Parallel** systems can execute multiple tasks at the same time, meaning that at a single isntance of time, there are > 1 tasks running.
- Concurrent systems can give the *illusion* of being parallel with rapid and frequent switching between threads, but this is not true parallelism.
- Nowadays, all hardware manufacturers support multiple hardware threads per core (modern Intel and AMD CPUs have 2 threads per core)

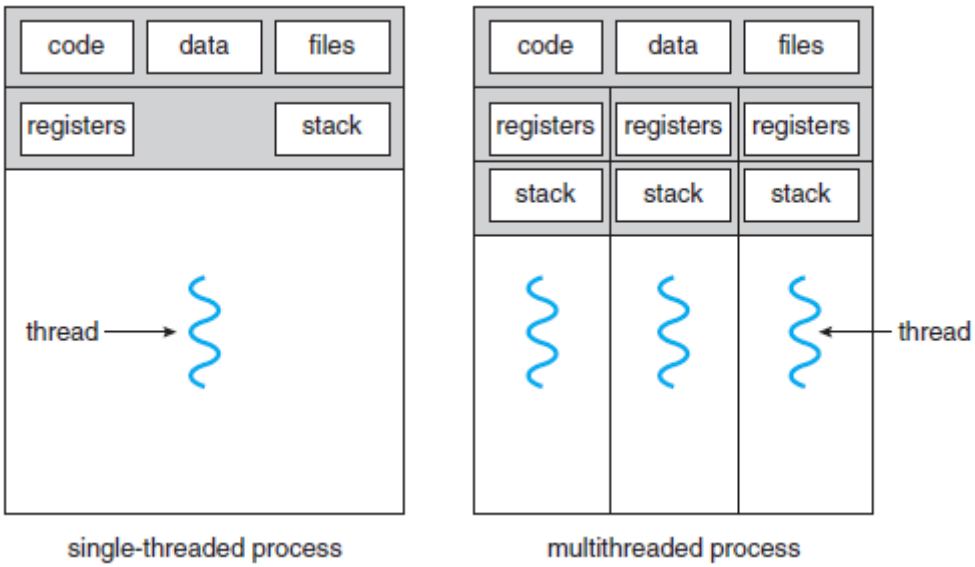

Figure 1: Single and Multi-Threaded Architectures

1.2.1 Amdahl's Law

- **Amdahl's Law** states that the the speedup obtained from running a program with a fraction of serial code given by S on N computational units (cores) is given by

$$speedup = \frac{1}{S + \frac{1-S}{N}} \tag{1}$$

- Amdahl's law proposes that the serial portion of the program has a disproportionate effect on any performance gain derived from adding more cores. (as $n \to \infty$, $speedup \to \frac{1}{S}$)
- Some argue that Amdahl's Law does not take into account the hardware performance enhancements used in the design of contemporary multicore systems.

1.3 Programming Challenges

- **Identifying tasks** that are simple and independent within the program logic.
- **Balanced** workloads on all tasks. If some tasks are less valuabe to the final output, then it may not be worth the effort to run them on separate cores.
- **Data splitting** to run on multiple cores
- **Data dependencies** between threads. Threads must be synchronized to avoid problems with multiple threads writing to/reading from the same memory location.
- **Testing and debugging** is more complicated due to many different execution paths possible.

1.4 Types of Parallelism

- **Data-level** parallelism involves distributing the data into multiple cores and each thread (running on a core) does the same task on this subset of the data
- **Task-level** parallelism involves different threads doing different tasks, on either the same data or different data.
- Real world applications use a hybrid of both types of parallelism.

2 Multithreading Models

These are different models of establishing a relationship between **user** and **kernel** threads.

2.1 Many-to-One Model

- Multiple user threads are mapped to a single kernel thread.
- Thread management is done by the thread library in user space, hence it is efficient.
- However if one thread executes a blocking system call, then all the the threads must stop execution.
- Also, multiple cores cannot be taken advantage of as only one kernel thread is present to access the kernel at a time, hence multiple user threads cannot run at the same time
- eg: Green threads on Solaris and early Java versions

2.2 One-to-One Model

- One user thread is mapped to one kernel thread
- Greater concurrency as another thread can be made to run when one thread makes a blocking system call. Multiple threads can also run in parallel on multiprocessors.
- Since creating a user thread also means creating a corresponding kernel thread, the overheads are high, and the OS can restrict the number of user threads being created.
- eg: Linux, Windows family OSes

2.3 Many-to-Many Model

- Multiple user threads are multiplexed to a smaller or equal number of kernel threads.
- The developer can create as many user threads as needed, and the corresponding kernel threads can run in parallel on multicore systems.
- If one thread does a blocking system call, another kernel thread can be scheduled to run.
- In a **two-level system**, many user threads are still multiplexed to a certain number of kernel threads, but user threads can also be **bound** to a kernel thread. (eg: Solaris <9)

3 Thread Scheduling

- On systems that support user-level and kernel-level threads, the kernel-level threads and not processes are scheduled by the operating system.
- In order to be scheduled, user threads need to be mapped to a kernel thread, although this mapping may not be direct but might be via a **lightweight process** (LWP).

3.1 Contention Scope

- On many-to-one and many-to-many implementations, the thread library schedules user threads onto available LWPs. This scheduling is called **Process Contention Scope** or PCS scheduling.
- PCS is called so because competition for CPU now happens between user threads scheduled on one process.
- To decide which kernel thread should be scheduled onto the CPU, **System Contention Scope** or SCS scheduling is used. SCS scheduling is used among all the threads in the system.
- PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run.

- User-level thread priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread.
- PCS will typically preempt the thread currently running in favor of a higher-priority thread; however, there is no guarantee of time slicing among threads of equal priority.

4 Thread Libraries

- Thread Libraries are APIs for developers to create and manage threads
- They may be implemented entirely at the user level (ie. the data structures and the code are all in user space) and no system calls are made.
- They may also be implemented with kernel support, ie. code and data structures are in kernel space, and function calls of this API involve system calls.
- The three main thread libraries in use today are `pthreads`, Windows Threads and the Java threading library (for applications running on the JVM).

4.1 `pthread` code example

The following is a code example that launches a separate thread to compute the sum of the first n natural numbers.

```c
#include <stdio.h>
#include <pthread.h>

void* runner(void* arg);
int sum = 0;

int main(int argc, char* argv[])
{
    pthread_attr_t attr;
    pthread_t tid;

    int num = atoi(argv[1]);

    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, (void*)&num);
    pthread_join(&tid);

    printf("Sum = %d\n", sum);
}

void* runner(void* arg)
{
    int n = *((int*)arg);
    for(int i=1; i<n; ++i)
        sum += i;
    pthread_exit(NULL);
}
```

`pthread` code example

- The `pthread_attr_t()` function is used to initialize the `pthread_attr_t` structure that holds the thread attributes.
- The `pthread_create()` function takes in the `pthread_t` structure, the attributes, the pointer to the function to be run on that thread (which must have signature of `void* function(void* arg)`) and the argument to be passed to the thread function.

- Inside the thread function, the `void*` is converted to `int` and the computation is done. At the end of the computation, the `pthread_exit(NULL)` function call signifies the end of that thread and return of control back to the main process.
- The `pthread_join(&tid)` function is used to make the main process wait for the child thread called `tid` to finish executing.

5 Process Synchronization

- Process synchronization is required to avoid **race conditions**.
- A race condition is a situation where multiple threads or processes access the same data concurrently, and the final outcome is dependent on the order in which the processes/threads accessed the data.
- Synchronization ensures that only one process can access the data at any instant in time

5.1 The Critical-Section Problem

- In a system consisting of n processes $\{P_0, P_1, P_2, ..., P_{n-1}\}$, each process has a region of code called the **critical section** where it performs operations like updating a global variables, updating a table or opening a file.
- The important feature of the critical section is that when one process is executing its critical section, no other process is allowed to execute its critical section. That is, no two processes may execute their critical sections at the same time.
- Each processes requests entry into its critical process in the **entry section**, executes the critical section, and exits using the **exit section**. All other code in the process falls under the **remainder section**.
- The problem statement is to design a protocol that the n processes can use to coordinate their execution.
- The solution to the critical-section problem has the following characteristics:
  - It must ensure **mutual exclusion**, ie. make sure that if process P_i is executing its critical section, then no other process is executing its own critical section.
  - It must ensure **progress**. If no process is executing its critical section, and there are some processes that wish to enter their critical sections, then only processes that are not executing their remainder sections are eligible to be selected to run their critical section next, and this selection cannot be postponed indefinitely.
  - In short, progress implies that if a process is not using the critical section, then it should not stop any other process from accessing it.
  - **Bounded waiting**. Between a process requesting access to the critical section, and the granting of the request, there is a finite number of times that all other processes can access their critical sections.
  - In other words, the waiting time for a process between request for access and actual access must not be infinite.
  - Read this link for clear explanation.
- No assumptions are made about the **relative speed** of the processes, but it is assumed that all processes are executing at non-zero speed.
- In kernel mode, there are many scopes for race conditions (eg: two processes simultaneously opening a file may cause a race condition in the file table maintained by the OS).
- With **preemptive kernels**, a process can be preempted when it is running in kernel mode. This sort of kernel can have race conditions, and is more prone to race condition in SMP (Symmetric MultiProcessing) systems, as more than one kernel process may run on different processors.

- In a **non-preemptive kernel**, a process has continuous control of the CPU until it blocks, waits or completes execution. This kernel design is free from race conditions as only one process is active in the kernel at a given instant of time.

5.2 Peterson's Solution

- This is a software-based solution to the critical-section problem.
- Peterson's Solution works for two processes P_1 and P_2 (which are represented as P_i and P_j where $j = 1 - i$), and it assumes that the `load` and `store` machine instructions are **atomic**, ie. cannot be interrupted.
- The two process share two data items, `bool flag[2]` and `int turn`.
- The `turn` variable indicates which process' turn it is to enter the critical section, if the `turn == i` then process process P_i is allowed to execute in the critical section. The `flag[2]` array indicates whether the process is ready to enter the critical section, so `flag[i]` is set to true when process P_i is ready to enter its critical section.
- To enter the critical section, process P_i sets the value of `flag[i]` to true, and then changes the value of `turn` to i.
- If both processes want to enter the critical section, then they both attempt to overwrite the `turn` variable, and only one edit survives (the other one takes place and then almost immediately gets overwritten), hence only that process gets permission to enter the critical section.

```
do{
     flag[0]=true;
     turn = 1;
     while(flag[1]==true && turn ==1)
         ; //Busy Waiting
     //Critical Section

     flag[0] = false;

     //Remainder Section
}while(true);
```

Code for Process P_0

```
do{
     flag[1]=true;
     turn = 0;
     while(flag[0]==true && turn ==0)
         ; //Busy Waiting
     //Critical Section

     flag[1] = false;

     //Remainder Section
 }while(true);
```

Code for Process P_1

5.2.1 Validity of Peterson's Solution

- The process P_i will be inside its critical section when `flag[j]` is false (meaning that P_j has left the critical section) or `turn == i` (meaning that P_j is waiting for its turn to enter the critical section).
- Hence, if both process are in their critical section, then this must imply that turn=0 and turn=1 are both true at the same time, which is not possible. Hence **mutual exclusion** is ensured.
- **Bounded waiting** is ensured as no process waits more than one turn to get control of its critical section.
- A process cannot immediately re-enter the critical section if the other process has set its flag to say that it would like to enter its critical section. Hence Peterson's Algorithm also ensures **progress**.
- As Peterson's Algorithm ensures all 3 conditions, it can be considered as a valid solution to the Critical Section Problem.

5.2.2 Drawbacks of Peterson's Solution

- The algorithm assumes that all load and store operations at the hardware level are *atomic*, ie. they cannot be interrupted. On modern multiprocessor systems this assumption does not hold.
- Peterson's Solution also is hard to implement for more than two processes, and when it is extended for more than 2 processes it fails to guarantee bounded waiting.

5.3 Hardware Synchronization: Locking

- In contrast to the software solution presented above, most modern systems expose APIs to application and kernel programmers which use the principle of **locking**.
- In non-preemptive kernels, the critical-section problem is solved by a primitive form of locking where no interrupt is allowed to occur while a shared variable is being modified (ie. some process is in its critical section).
- This is not a feasible solution for multiprocessor systems as it reduces the performance and can cause timing issues with the CPU clock (while relies on interrupts to refresh itself).
- In most modern hardware platforms, the `test_and_set()` and `compare_and_swap()` instructions are provided as **atomic** (ie. non-interruptable) instructions.

5.3.1 Test and Set

- The test-and-set instruction is implemented as below

```
bool test_and_set(bool* target){
    bool return_value = *target;
    *target = true;
    return return_value;
}
```

`test_and_set()` implementation

- This instruction is an **atomic** instruction, meaning that if two test-and-set instructions are executed simulatenously on different CPUs, then they will be executed one after the other in some arbitrary order.
- Mutual exclusion can be implemented using the test-and-set instruction as shown below. The shared Boolean variable `lock` has an initial value of `false`.

```
do{
    while(test_and_set(&lock))
        ; //Do nothing
    //Critical Section
    lock = false;
    //Remainder Section
}while(true);
```

Process structure using `test_and_set()`

5.3.2 Compare and Swap

- The compare-and-swap instruction is implemented as below

- The compare-and-swap instruction compares the value present at the location of `value` and then changes the value to `new_val` only if the value matches the expected value.

```
int compare_and_swap(int* value, int expected, int new_val){
    int temp = *value;
    if(temp==expected)
        *value = new_val;
    return temp;
}
```

compare_and_swap() implementation

```
do{
    while(compare_and_swap(&lock, 0, 1) != 0)
        ;  //Do nothing
    //Critical Section
    lock = 0;
    //Remainder Section
}while(true);
```

Process structure using `compare_and_swap()`

- The process is implemented as below
- The above algorithms satisfy mutual exclusion but do not satisfy bounded waiting.
- For this a new solution is presented that satisfies all the three conditions, and maintains the variables `boolean lock;` and `boolean waiting[n];` where n is the number of processes.
- This implementation is given below

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

/* Critical Section */

j = (i+1)%n;
while(j != i && !waiting[j])
    j = (j+1)%n;
if(j==i)
    lock = false;
else
    waiting[i] = false;

/* Remainder Section */
}while(true);
```

Bounded Waiting mutual exclusion with `test_and_set()`

- The variables `waiting[n]` and `lock` are both `false` initially.
- As this hardware-based solution is not accessible easily to application and kernel developers, most OSes provide atomic routines to manage locks.

6 Mutex Locks

- Mutex locks are software solutions to the critical-section problem that use two functions named as `acquire()` and `release()`, which set and release the lock.

- The definitions of these functions are as below.

```
void acquire()
{
     while(!available)
         ; //Busy waiting
     available = false;
}
```

Code for `acquire()` function

```
void release()
{
      available = true;
}
```

Code for `release()` function

- Both functions manipulate a shared boolean variable called `available` which indicates the availability of the mutex lock for the process. The process structure with mutex locks is as follows

```
do{
    acquire();
    //Critical Section
    release();
    //Remainder Section
}while(true);
```

- The disadvantage of this solution is that it requires **busy waiting**. When one process sets the lock, all the other processes that wish to access their critical sections are forced to loop without any useful work being done.
- Such a locking mechanism is also called a **spinlock** as a process that is not allowed access to the critical section essentially *spins* in place until it is allowed to.
- Busy waiting is especially a problem in multiprogramming systems where a single CPU is shared between multiple processes and where CPU cycles are wasted doing no useful work due to the lock not being available.
- Spinlocks offer the advantage, however, of not having to undertake a context switch everytime a process needs to wait, which can be tolerated when locks are to be held for a short time.
- Spinlocks are useful on multiprocessor systems where one thread can spin on one CPU while another threads keeps on running.

7 Semaphores

- Semaphores provide more sophisticated process synchronization mechanisms than simple mutex locks.
- A semaphore `S` is an integer variable that is accessed through the two atomic functions `wait()` and `signal()`.
- Both these operations must be **atomic** and indivisible (ie. no other process can modify the value of `S`)
- There are 2 main types of semaphores: **binary** and **counting** semaphores.
- In a binary semaphore, the integer `S` can take only values of 0 and 1. When the initial value of `S` is 1, this is identical to a mutex lock.
- In a counting semaphore, the integer `S` can take any integer value. The initial value is set to the number of resources avaiable. Everytime a process needs access to a resource, it uses the `wait()` call, which decrements S, and everytime the resource is released by the process, the `signal()` call is used to increment S. When S reaches 0, no more resources are available, and any process which requests a resource at this point must undergo busy waiting.

```
void wait(S){
    while(S<=0)
        ; //Busy waiting
    S--;
}
```

Code for `wait()` function

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

Code for `signal()` function

- Semaphores can be used to restrict the order of statement execution between 2 processes.
- Suppose statement S_1 in process P_1 must execute before statement S_2 in process P_2. A common semaphore called `synch` is shared between P_1 and P_2 and its initial value is **zero**. The structure of P_1 and P_2 is as shown below.

```
S1;
signal(synch);
```

Code for Process P_1

```
wait(synch);
S2;
```

Code for Process P_2

7.1 Semaphore Implementation

- The wait operation is modified to remove the need for busy waiting.
- When the value of the semaphore is not positive, the process blocks itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.
- When one process does a `signal()` operation, a `wakeup()` routine is needed to remove a process from the waiting queue to the ready queue.
- The new semaphore definition is as follows:

```
typedef struct{
    int value;            //Semaphore value
    struct process* list; //Array of processes
}semaphore;
```

The `wait()` and `signal()` calls are implemented as follows. Now they accept a single pointer to the shared instance of the structure `semaphore`.

- The semaphore variable `value` may take a negative value, this is due to the switching of the order of the checking and increment/decrement operations in the `wait()` function.
- If the value is negative, the magnitude is the number of processes that are waiting for access to resources.
- In SMP systems, alternate locking mechanisms (using `compare_and_swap` are used to ensure that the wait and signal are executed atomically.
- The busy waiting has not entirely been eliminated in this solution, but it has been moved to the critical section from the entry section. In a properly written application with a short critical section (that is rarely executed), this is advantageous over the earlier solutions

```
void wait(semaphore* S)
{
    (S->value)--;
    if(S->value < 0)
    {
        add(current_process, S->list);
        block();
    }
}

void signal(semaphore* S)
{
    (S->value)++;
    if(S->value <= 0)
    {
        remove(P, S->list); //Remove process P from S->list
        wakeup(P);          //Change status of P from waiting to ready
    }
}
```

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Figure 2: Deadlock illustration

8 Deadlocks and Priority Inversion

8.1 Deadlocks

- A **deadlock** is a situation in which 2 or more processes are waiting indefinitely for an event to occur that can only be executed by one of the processes.

- Suppose that P_0 executes `wait(S)` and then P_1 executes `wait(Q)`.When P_0 executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`.
- Similarly, when P_1 executes `wait(S)`, it must wait until P_0 executes `signal(S)`. Since these `signal()` operations cannot be executed, P_0 and P_1 are deadlocked.

8.2 Priority Inversion

- The situation of **priority inversion** arises when a higher priority process' access to shared resources is blocked by a lower priority process that holds a lock on the shared resource.
- For example, if there are three processes L, M and H with priority order $L < M < H$ and L has a lock on resource R. If H desires the resource R then it must wait for L to release the lock.
- If L is preempted by M before it can release its lock on R then H is made to wait even longer.

8.2.1 Priority Inheritance Protocol

- This is one possible solution to the Priority Inversion problem.

- All processes that access a shared resource needed by a higher priority process inherit this higher priority until they release their lock on this resource.
- Once the lock is released, the priority returns to its normal (lower value).
- In the above example, L would inherit the priority of H, hence it would not be preempted by M. Hence, after L releases the lock and returns to its original priority, H is the next process to access the shared resource (not M).

9 Classic Problems in Synchronization

9.1 The Bounded-Buffer Problem

- The producer and consumer processes share the following resources:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```

- It is assumed that the pool consists of n buffers each of size 1.
- The semaphore variables `empty` and `full` count the number of empty and full buffers respectively.
- The code for the producer and consumer is given below

```
do{
     next_produced = produce_item();

     wait(empty);
     wait(mutex);

     add(next_produced, buffer);

     signal(mutex);
     signal(full);
}while(true);
```

Code for producer process

```
do{
     wait(full);
     wait(mutex);

     next_consumed = remove_item(buffer);

     signal(mutex);
     signal(empty);

     /* Consume item in next_consumed */
 }while(true);
```

Code for consumer process

9.2 The Readers-Writers Problem

- Processes are divided into 2 categories: those that only read from a shared memory (readers), and those that only write to the shared memory (writers).
- The problems arise when readers and writers concurrently try to operate on the same memory, which can lead to the writer overwriting the data which was needed by the writer, or the reader reading an out-of-date value from the memory.
- The readers-writers problem has many variations, in particular the *first* and *second* readers-writers problems, which deal with priority order between readers and writers.
- In the **first** readers-writers problem, no reader must be kept waiting unless a writer has already obtained permission to access the shared memory.
- This implies that no reader should wait for other readers to finish simply because a writer is waiting

- In the **second** readers-writers problem, once a writer is ready, it should perform the write as soon as possible, without any readers interfering in between.
- This implies that if a writer is waiting to access the object, no new readers may start reading.
- While solving the first problem, writers should not starve. While solving the second problem, readers should not starve.

9.2.1 Solution to First Readers-Writers Problem

- The reader and writer share the common resources:

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

- The `mutex` semaphore is used to ensure mutual exclusion when the variable `read_count` is updated.
- The `read_count` variable keeps track of how many processes are currently reading the object.
- The semaphore `rw_mutex` functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

```
do{
     wait(mutex);
     read_count++;
     if(read_count==1)
         wait(rw_mutex);
     signal(mutex);

     /*Perform read operation*/

     wait(mutex);
     read_count--;
     if(read_count==0)
         signal(rw_mutex);
     signal(mutex);
}while(true);
```

Code for reader process

```
do{
     wait(rw_mutex);

     /* Perform write operation */

     signal(rw_mutex);
 }while(true);
```

Code for writer process

- If a writer is in its critical section and n readers are queued, then one reader is queued on `rw_mutex` while $n-1$ readers are queued on the `mutex` semaphore.
- When a writer executes the `signal(rw_mutex)` call, the next executed process may be a single waiting writer or any one of the n waiting readers. This decision is made by the scheduler.
- A more general solution of the readers-writers problem involves the use of special *reader-writer* locks in place of mutex/semaphore locks as above.
- Readers acquire such a lock in *read* mode, while writers acquire it in *write* mode.
- Multiple readers can acquire a lock in read mode, but only one writer can acquire a lock in write mode as writing needs mutual exclusion.

9.3 The Dining Philosopher's Problem

- There are n philosophers sitting around a table, who spend their entire lives eating or thinking. There are n chopsticks on the table, one between every two philosophers. There is one single bowl of food at the center of the table to be shared by all the n philosophers.
- A philosopher may pick up only one chopstick at a time. When a hungry philosopher has both chopsticks in hand, they start eating until they are full, without releasing the chopsticks. Once the philosopher is finished eating, they release both chopsticks and resume thinking.
- The problem statement is to devise a deadlock and starvation free manner in which the philosophers (labelled $1..n$) can pick up the chopsticks and eat.

9.3.1 Solution to Dining Philosopher's Problem

- The simplest solution is to represent the n chopsticks as an array of semaphores. The act of picking up a chopstick is represented as a `wait()` and the act of putting it down is represented as `signal()`.
- The structure of the ith philosopher is described below:

```
do{
    wait(chopstick[i]);
    wait(chopstick[(i+1)%n];

    /* Eating */

    signal(chopstick[i]);
    signal(chopstick[(i+1)%n];

    /* Thinking */

}while(true);
```

Process structure for Dining Philosopher's Problem

- This solution ensures that no 2 neighbours are eating at once, but if all n philosophers are hungry at once and all pick up the chopstick to their left then they will be stuck in a **deadlock** waiting for the neighbour to release their chopstick.
- Some possible remedies to the deadlock problem are:
  - Allow a philosopher to eat only if both chopsticks are avaiable. This turns the eating into a critical section.
  - Using an asymmetric solution where an odd-numbered philosopher first picks up the left then the right chopstick, but the even-numbered philosopher first picks up the right then the left chopstick.

10 Examples of Process Synchronization

10.1 Windows Synchronization

- On a single processor system, the Windows kernel temporarily masks all interrupts that can access shared resources.
- On multiprocessor systems, shared resources are protected by the kernel using spinlocks (only for short code segments). The kernel ensures that threads will never be preempted while they hold a spinlock.
- Outside of the kernel, Windows uses **dispatcher objects** for synchronization.

- Dispatcher objects provide many different methods for synchronization such as mutexes, semaphores, events (condition variables) and timers (used to notify one or more threads that a certain time has elapsed).
- Dispatcher objects may be in **signaled** state or **nonsignaled** state. In signaled state, a thread will not block while taking control of the object, while in nonsignaled state, threads will block while attempting to take control of the dispatcher object.
- When a thread blocks on a nonsignaled dispatcher object, it is placed into the waiting queue from the ready queue.
- When the dispatcher object moves to signaled state, the kernel checks if any threads are waiting for it. If there are such threads, they are moved from the waiting to the ready queue
- Windows also provides **critical-section objects** which are user-mode mutexes.
- On a multiprocessor system, a critical-section object first uses a spinlock while waiting for the other thread to release the object. If it spins too long, the acquiring thread will then allocate a kernel mutex and yield its CPU.
- Critical-section objects are particularly efficient because the kernel mutex is allocated only when there is contention for the object. In practice, there is very little contention, so the savings are significant.

10.2 Linux Synchronization

- The simplest synchronization technique in Linux is the use of atomic data types that allow operations (for example math operations on integers) to be done entirely atomically.
- Linux also provides spinlocks, mutex locks and semaphores as mechanisms on multi-processor systems
- On single processor machines (eg: embedded systems), Linux allows disabling and enabling kernel preemption in place of acquiring and releasing a lock. This is done using two system calls `preempt_disable()` and `preempt_enable()`
- Every process contains a `thread_info` structure which contains one counter (called `preempt_count`) for the number of locks being held by the task, which is incremented or decremented as a lock is acquired or released respectively.
- If `preempt_count` is greater than 0, then it is not safe for the task running to be preempted, but if it is 0 then the task can be safely preempted.

10.3 pthread Synchronization

- The `pthread` API provides a user-accessible implementation of synchronization structures independent of the kernel for POSIX-compliant systems.
- Mutex locks are provided using the `pthread_mutex_t` data structure and initialized using the `pthread_mutex_init()` function.
- They can be acquired and released using the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions respectively.
- Semaphores (part of a separate semaphore API) are declared using the `sem_t` data structure, and initialized using the `sem_init()` function.
- The wait and signal operations of semaphore are implemented as the `sem_wait()` and `sem_post()` functions respectively.
- The semaphore API provides **named** and **unnamed** semaphores. The difference is that named semaphores have file system names and can be accessed across multiple unrelated processes. Unnamed semaphores can only be accessed by threads belonging to one single process.

11 Deadlocks

11.1 System Model

- The set of available resources in the system is modelled as belonging to different classes.
- In each class of resources $R_i, i \in \{1, 2, ..., n\}$, there are W_i instances, all of which are **entirely identical**.
- A process undertakes resource utilization in the following order
  1. **Request** for access. If not immediately available then wait
  2. **Use** the resource
  3. **Release** the resource for use by others

11.2 Necessary Conditions

The follwing conditions must simultaneously hold for a situation to be characterized as a deadlock.

- **Mutual Exclusion**: At least one resource is held by a process and cannot be shared with other processes (other processes have to wait for it to be released)
- **Hold and Wait**: A process that currently holds resource R_1 is waiting for another resource R_2 that is being currently held by some other process.
- **No Preemption**: Resources can only be released if the process voluntarily releases them itself. No external process (not even the kernel) can preempt the process and force it to give up the lock on the currently held resource.
- **Circular Waiting**: There exists a set of process $P_1, P_2, ..., P_n$ such that P_1 is waiting for a resource held by P_2, P_2 is waiting for a resource held by P_3, P_3 is waiting for a resource held by P_4, and so on, and P_n is waiting for a resource held by P_1.

11.3 Resource Allocation Graph

- The allocation of resources to processes is visualized by a graph.
- The vertices are of two types, the active processes (circles) and the resource classes in the system (rectangles). Instances of the same resource class are denoted as dots within the rectangle of that resource class.
- There are two types of edges, the request edge (directed from process P_i to resource class R_j) and the assignment edge (directed from resource class R_j to process P_i).
- If the graph contains no cycles, then there are no deadlocks in the system.
- If the graph contains a cycle, there *may be* a deadlock.
- If all resource types have only one instance, then a cycle implies a deadlock has occurred. In this case, the existence of a cycle is necessary and sufficient condition for deadlock to exist.
- But if resource types in the cycle have multiple instances, then cycle does not necessarily imply a deadlock.

11.4 Methods for Handling Deadlocks

- The following methods may be used to avoid deadlocks:
  - Deadlock prevention (preventing the system from entering a deadlocked state under any circumstance)
  - Allow the system to enter a deadlock state, detect it and then recover from it.
  - Ignore the existence of deadlocks and continue normal operation (used in Windows and Linux). This places the responsibility of deadlock avoidance on the application programmer.

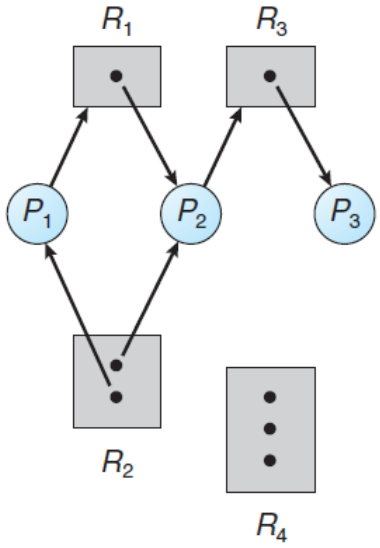

Figure 3: Resource Allocation Graph

12 Deadlock Detection

- In a system where each resource class has only one instance, deadlocks are detected from the **wait graph**.
- The wait graph is constructed from the resource allocation graph, where each vertex denotes a process and an edge from P_i to P_j denotes that P_i is waiting for P_j.
- A cycle in the wait graph denotes the existence of a deadlock, and detecting a cycle in the graph is an $O(n^2)$ process where n is the number of processes (ie. vertices) in the graph.

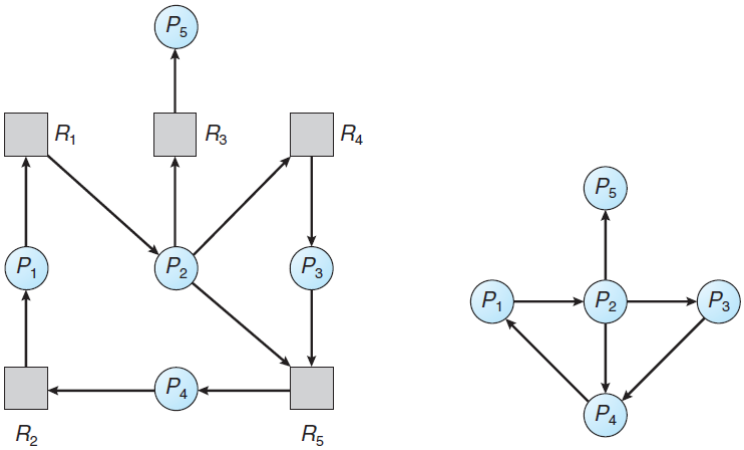

Figure 4: Resource Allocation Graph and the corresponding Wait Graph

12.1 Multiple Instance per Resource Class

- Let there be m resource classes and n active processes in the system.
- The algorithm used to detect a deadlock in such a situation makes use of the following data structures:

- **Available**: A vector of length m indicating the number of resources available per resource class.
- **Allocation**: An $n \times m$ matrix which denotes the number of instances of each resource class allocated to each process.
- **Request**: An $n \times m$ matrix which denotes a request made by a process for a resource instance. If $R[i][j] == k$ then process R_i is requesting k instances of resource class R_j.

12.1.1 Algorithm for Deadlock Detection

1. Let `work` and `finish` be vectors of length m and n respectively. Initialize `work = available`. For $i \in [0, n-1]$ if `allocation[i]` $\neq 0$, `finsh[i]=false` else `true`.
2. Find an index `i` such that `finish[i] == false` and `request[i]` $\leq$ `work`. If no such `i` exists, go to step 4
3. Set `work = work + allocation` and `finish[i] = true`. Go to step 2
4. If `finish[i] == false` for some $i \in [0, n-1]$, then system is in a deadlocked state. Process P_i is in a deadlock.

13 Deadlock Prevention

13.1 Mutual Exclusion

- Some files (such as read-only files) do not require any mutual exclusion privileges
- But other resources may require control to ensure no race conditions,

13.2 Hold-and-Wait

- To ensure that hold and wait does not occur in the system, the OS must make sure that when a process requests for a resource, it is not holding any other resources.
- One such protocol to implement this is the process requesting for all of its resources at the very beginning and getting all of them at that time.
- This is done by forcing all system calls for resource allocation to come before any other system calls.
- Another protocol is to allocate resources to a process only if that process is not holding any other resources. If it is holding any other resources it must release them before asking for more resources.
- Both protocols here suffer from the problem of starvation (processes asking for popular resources may have to wait infinitely) and low resource utilization (due to resources being allocated and not used by processes)

13.3 No Preemption

- If a process holding some resources asks for another resource which is currently not available, then all the currently held resources may be released and added to the list of resources requested by that process
- Then the process must wait until both old and new resources are available for the process to acquire.
- An alternate approach is to allocate resources to a process P_1 that asks for them if they are available.
- If the resources are being held by another process P_2 that is, in turn, requesting for another resource, then we preempt the desired resources from the waiting process P_2 and allocate them to the requesting process P_1.

- If the resources are neither available nor held by P_1, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them
- The second protocol is applied to resources that can be saved and restored later (like CPU registers and memory), but not to resources like mutex locks/semaphores.

13.4 Circular Waiting

- Let there be the resource classes $\{R_1, R_2, ..., R_m\}$ each with its own unique integer value, given by the function $f(R_i)$.
- A process can only request the resources if the requests are made in an increasing order of this function f. For example, a request for resource class R_j can be made after a request for class R_i only if $f(R_i) < f(R_j)$
- Alternatively, a process that requests a resource class R_j must free all instances of resource class R_i such that $f(R_j) \geq f(R_i)$
- All the applications written by third party programmers must also conform to this ordering to avoid deadlocks.
- The function f must be chosen in such a way that it follows the real-world usage order of system resources.
- Deadlocks can be prevented by using system utilities like **witness** (on BSD UNIX systems) that determines the order of locks taken in the system and checks for a deadlock possibility if locks are held out of order.