Process Management

- To implement the process model, the OS maintains a table, an array of structures, called the **process table**.
- It is also called **Process Control Blocks (PCB)**.
- Stores one entry per process.
- Each entry contains important information about a process; process ID, process state, its program counter, stack pointer, memory allocation, scheduling information, …….everything about a process when the process goes from running to ready/blocked state. Figure 2.4.
- The various fields of the PCB are highly system dependent, not fixed.
- Figure 2.5 shows the steps what does an OS do when an interrupt occurs.
- **CPU Utilization in Multiprogramming:** Suppose that a process spends a fraction of p of its time waiting for I/O complete. If there are n processes in the memory at once, then the probability that all the n processes are waiting for the I/O is p^n. So, the CPU utilization is calculated by

  $$CPU\ utilization = 1 - p^n$$

- **Degree of Multiprogramming:** It is the maximum number of processes that a single-processor system can handle efficiently. the primary factor affecting the degree of multiprogramming is the amount of memory available to be allocated to the executing processes. CPU utilization, in multiprogramming, gets increased with the increased size of memory.

All the information of the lectures (CSE 353) are prepared from books, research articles, and online resources.

Process Management

Thread: A portion/segment/subset of a process.

- A process may have multiple threads.
- Miniprocesses within a process.
- In an application, there may be multiple activities that go on at once. So those are divided into threads **who run in parallel**.
- Used to improve the processing of an application.
- Faster execution.
- Easier to create and destroy a thread than a process as it is lighter weight than a process.
- Threads are truly useful on systems with multiple processors.
- For example, consider a spreadsheet application.

Process Management

- Another example of thread: a web server.
  - Dispatcher thread reads incoming request for a web page from a client through the network.
  - The dispatcher thread, after examining the request, wakes up a worker thread, from blocked to ready state and hands the request to it.
  - Now, the worker thread checks if the requested page is avai -lable in the web page cache. If the page is found, the worker thread returns the page to the client and waits for a new request. If the page is not found, it issues a read operation to get the page from the disk and stays blocked until the disk operation is performed.
  - During this blocked time, another thread may be chosen to perform other work.
  - Figure 2.9.

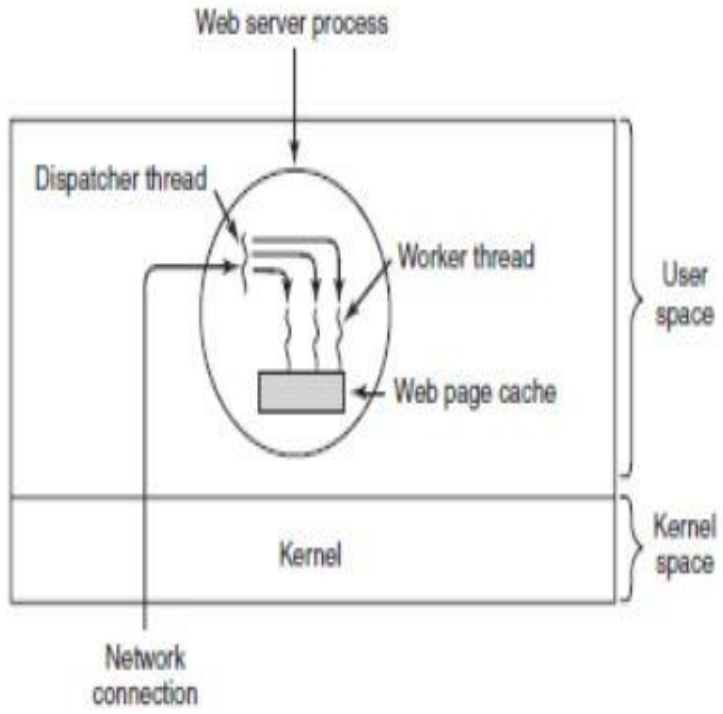

Figure 2-8. A multithreaded Web server.

Process Management

- What happens with the web server in absence of the multiple threads? (single threaded web server)
- So, multithreading is the situation of allowing multiple threads in the same process.
- The CPU switches rapidly back and forth among the threads providing the illusion that the threads are running in parallel.
- Different threads in a process are not independent as different processes.
- All the threads share the same address space, that means, they also share same global variables.
- thread_create (), thread_join(), thread_exit(), thread_yield(),……..

Process Management

- **Pthread/POSIX Threads**:
  - Portable threads, a standard for threads defined by IEEE, 1003.1c.
  - This standard defines more than 60 function calls.
  - Most Unix systems support it.
  - All Pthreads have certain properties: an identifier, a set of registers, program counter, stack, scheduling parameters,
  - Implemented in C using pthread.h header file and a thread library.
  - Figure 2.15

Process Management

Implementing Threads in User Space:

- The thread package is implemented entirely in user space.
- The kernel knows nothing about this.
- In this case, threads are **implemented by a library;** PThreads, Java Threads, Windows Threads.
- Here each process owns a **private thread table** to keep track of the threads.
- The **main advantage** is that this thread package can be implemented on an OS that does not support threads.
- The kernel is not aware of the existence of these threads, it handles them as if they were single threaded process.
- The **other advantage** is that each process has its own customized scheduling algorithm.
- When a thread is moved to ready state or blocked state, the information needed to restart the thread is stored in the thread table, exactly the same way as the kernel stores information about processes in the process table.
- The **disadvantage** is that the entire process is blocked if one user-level thread performs blocking operation.

Process Management

Implementing Threads in the Kernel:

- The kernel has **a thread table** that keeps track of **all the threads** in the system.
- The thread table holds each thread's registers, state, and other information.
- When a thread wants to create anew thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction of a thread by updating the kernel thread table.
- When a thread blocks, the kernel can either run another thread from the same process if it is ready, or a thread from a different process.
- There is no thread library but an API to the kernel thread facility.
- Multiple threads of the same process can be scheduled on different processors in kernel-level threads.
- **Kernel-level threads are slower** than user-level threads.
- **Thread recycling** is done in this approach. When a thread is destroyed, it is marked as not runnable, but its data structures are not affected. Later, when a new thread is created, an old thread is reactivated saving some overhead.

Process Management

Hybrid Implementation:

- Combines the advantages of user level and kernel level threads.
- One way is use kernel-level threads and then multiplex user-level threads onto some or all of them.
- The programmer can determine how many kernel threads to use and how many user-level threads to multiplex on each one.
- the kernel is aware of only the kernel-level threads and schedules those.

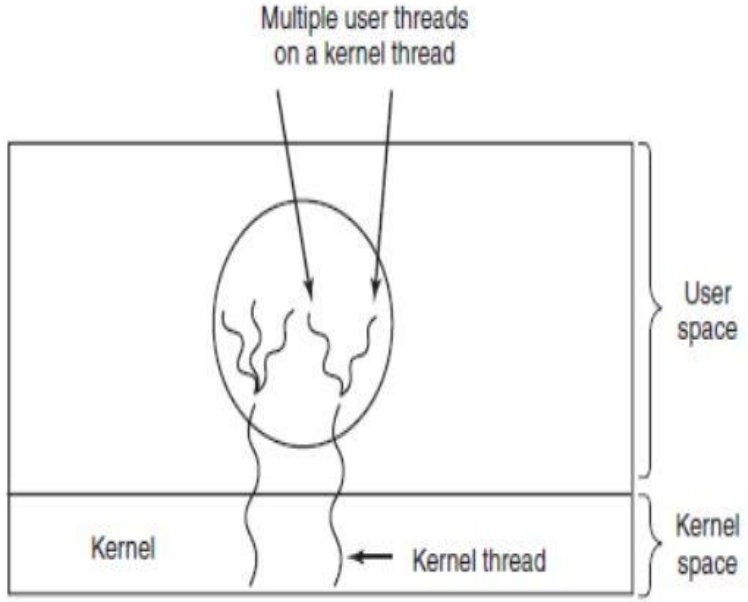

Figure 2-17. Multiplexing user-level threads onto kernel-level threads.

Process Management

Inter Process Communication:

- Processes frequently communicated with each other for sending output of one process to another.
- So, there is a need for communication between processes in a well-structured way.
- Three issues are concerned here:
  - How one process can pass information to another?
  - How to make sure two or more processes do not get into each others way?
  - Proper sequencing when dependencies are present.
- **Inter Process Communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions.**
- Processes can communicate with each other through **shared memory** and **message passing**.

Process Management

- **Race condition**
  - The processes that work together can share some **common storage space in the main memory** or **a shared file**.
  - May occur if commands to read and write a large amount of data are received at almost the same instant, and the machine attempts to overwrite some or all of the old data while that old data is still being read.
  - So, it is a situation when two or more processes are reading and writing some shared data and the final result depends on who runs precisely when. It may happen, when a process starts using a shared variable before the other process was finished with that.
  - Debugging program with race condition is not easy.
- **Critical region**
  - To avoid race condition, there must be some way to prohibit more than one process from reading or writing the shared data at the same time.
  - **The part of the program where shared memory is accessed is called the critical region or critical section.**
  - Race condition could be avoided if it could be managed that no two processes would be in the critical region at the same time. Here, the following conditions need to be remembered:
    - No two processes would be in the critical region at the same time.
    - No assumptions may be made about the speed or the number of CPUs.
    - No process running outside the critical region may block any process.
    - No process should have to wait forever to enter its critical region.

Process Management

- **Mutual Exclusion**
  - ➢Is a way to make sure that if one process is using a shared variable or file, the other process will be excluded from doing the same thing.

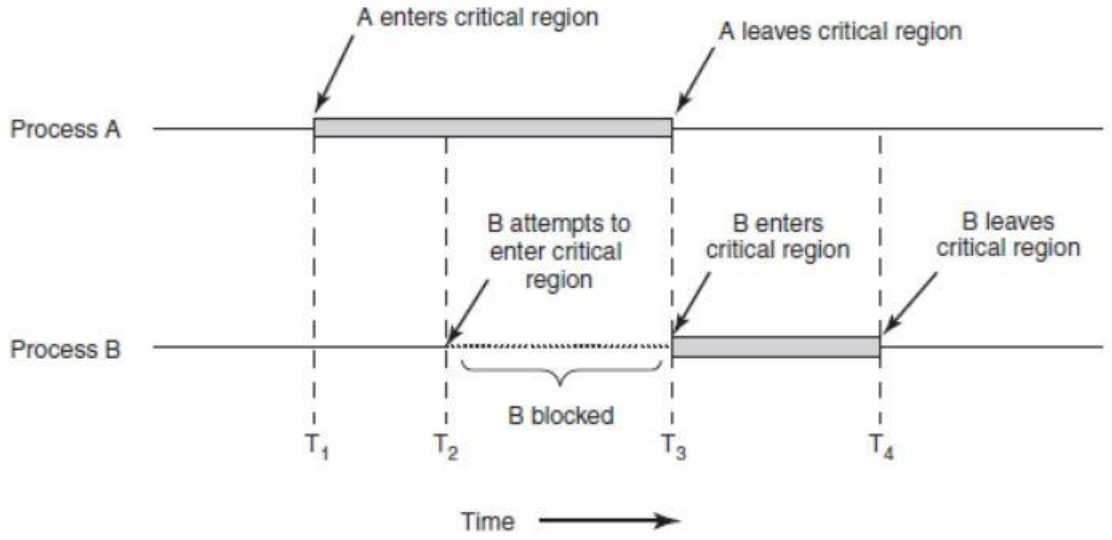

Figure 2-22. Mutual exclusion using critical regions.

Process Management

- **Achieving Mutual Exclusion:**
  - **Disabling Interrupts**
    - The simplest solution is to have each process turn off all interrupts after entering the critical region and re-enable them just before leaving the critical region. If the interrupts are disabled, then the CPU will not be switched to other process.
    - But **it is not wise to give user process the power to turn off interrupt**. Because, it might happen that a process never turns them on again.
    - In a multiprocessor system, the disabling interrupts affect the respective CPU only that executes that disable instruction; other CPUs will continue running and can access the shared memory.
    - It is convenient to disable interrupts by the kernel itself for a few instructions while it is updating variables, files or data. It is a useful process.

Process Management

- **Lock Variables**
  - A software solution.
  - There will be a single shared variable Lock with initial value 0.
  - When a process wants to enter its critical region, it checks the lock first.
  - If it is 0, the process sets it to 1 and enters the critical region.
  - If it is already 1, the process will wait until it becomes 0.
  - Again, it causes a fatal error. Suppose, one process reads the Lock 0. Before, it sets Lock to 1, another process is scheduled and sets the Lock 1 and enters the critical region. And the first process also sets the Lock 1 and enters the critical region. Result is two processes are in the critical region at the same time.

Process Management

- **Busy Waiting**
  - Here, a Lock variable, called turn, keeps track of whose turn it is to enter the critical region and examine or update the shared memory.
  - Continuously testing a variable until some value appears is called **busy waiting**.
  - A lock that uses busy waiting is called **lock spin.**

  - ```
    while(true) {  // for process 0 figure 2.23
            while(turn!=0);
            critical_region();
            turn=1;
           noncritical_region ();
     }
    ```
    ```

Process Management

- **Peterson's Solution/Algorithm**
  - This solution consists of two procedures.

```
#define FALSE 0
#define TRUE 1
#define N 2

int turn; int interested[N];
void enter_region(int process) {
    int other;
    other=1-process;
    interested[process]=TRUE;
    turn=process;
    while(turn==process && interseted[other]==TRUE);
}
void leave_region(int process) {
          interested[process]= FALSE;
}
```

  - Peterson's solution keeps a process waiting in the loop until its turn comes. It not only wastes CPU time, but also creates unexpected effects.; **priority inversion problem**.

Process Management

- **Sleep and Wakeup**
  - The producer-consumer problem/the bounded buffer problem as an example of sleep and wakeup method.
  - In this approach, two system calls **sleep** and **wakeup** are used to block and wakeup a process by another process.
  - Here, two processes share a fixed size buffer.
  - The producer puts information in the buffer, and the consumer takes it out.
  - Problem: If the producer wants to put new information in the buffer, but it is already full. Similarly, if the consumer wants to take out a new information, but the buffer is empty. So, both producer and consumer go to sleep mode until get waken up.
  - A **variable count** is used to keep track of the number of items in the buffer.
  - Yet a problem is there. Wakeup bit can be used.

```
#define N 10
int count=0;
void producer(){
    int item;
    while(true){
       if(count==N) sleep();
       insert_item(item);
       count++;
       if(count==1) wake_up(consumer);
    }
}
```

```
void consumer(){
    int item;
    while(true){
       if(count==0) sleep();
       item=remove_item();
       count--;
       if(count==N-1)
            wake_up(producer);
    }
}
```

Process Management

- **Semaphore**
  - Was proposed by Dijkstra in 1965 which is a very significant technique using a simple integer value.
  - Semaphore is simply **an integer variable** that is shared between processes.
  - This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.
  - **Semaphore is used to count the number of wakeups saved for future use.**
  - If the value is 0, indicating no wakeups were saved, or some positive value if one or more wakeups were pending.
  - There are two operations on the semaphore: **down and up** or **sleep or wakeup** respectively.
  - The down operation checks to see if the value is greater than 0, if so, it decrements the value, i.e. uses up one stored wakeup, and continues.
  - If the value is 0, the process is put to sleep, without completing the down for the moment.
  - The up operation increments the value of the semaphore addressed.
  - This solves synchronization problem and race condition.

Process Management

Process Scheduling

- When more than one process or thread are simultaneously at the ready state, they compute for the CPU at the same time.
- The part of the OS that makes the decision of which process to execute next is called the **scheduler** and the decision making algorithm is called the **scheduling algorithm**.
- In **Non-preemptive scheduling**, the CPU is allocated to the process till it terminates or switches to the waiting/block state.
- In **Preemptive scheduling**, the CPU is allocated to the processes for a limited time/fixed time. If it is still running at the end of the time, the scheduler suspends it and picks another process to run. It requires a **clock interrupt** occurs at end of the time.

Process Management

Scheduling Environments

- Batch
- Interactive
- Real time

Basic Terms

- **CPU burst** is the amount of time a process uses the CPU until it starts waiting for input or interrupted by another process.
- **Throughput** is the number of processes completed per unit time.
- **Turnaround time** is the time required for a particular process to complete, from submission time to completion.
- **Response time** is the time taken in an interactive program from the issuance of a command to the commence of a response to that command.

Process Management

First Come First Served Scheduling

- The simplest scheduling algorithm.
- Processes are assigned to the CPU in the order as they request it.
- There is a single queue of ready processes.
- When the running process blocks, the next process in the queue is run.
- When a blocked process becomes ready, it is added at the end of the queue as a new process.
- It is a non-preemptive scheduling.
- FCFS is a **Batch algorithm.**
- Easy to understand.
- Waiting time is not optimal.

Process Management

Shortest Job First Algorithm

- Another non-preemptive batch algorithm.
- Assumes the runtime is known in advance.
- The scheduler picks the shortest job first.

Figure 2-41. An example of shortest-job-first scheduling. (a) Running four jobs in the original order. (b) Running them in shortest job first order.