Lecture Notes in Computer Science:
Producer-Consumer Problem in Process Synchronization

Truc Huynh[1]

[1] Computer Science Department, Indiana University-Purdue University
Fort Wayne, IN, 46805,

huyntl02@students.pfw.edu

Abstract. Summary of the functionality of process and threads in Operating System. The architecture of process, thread, Process-Scheduler, Concurrency, and Synchronization. Also, the research gives an overview of multithreading, and compare multi-process architecture with multi-threads architecture. Finally, the research explains what process synchronization and its issues. I use the Producer/Consumer problem (Bound-Buffer Problem) to demonstrate synchronization problems.

Keywords (*)**:**

- **Application:** are reside on the hard drive of a computer [5].
- **Code:** Programming instruction that is going to be executed on a CPU [5].
- **Context Switch:** The act of stopping a running and start a new one is called a context switch [5].
- **Data (Heap):** contain all the data that the application need [5].
- **Instruction Pointer:** Address of the next instruction to execute [5].
- **Main Thread:** Each process contains at least one main Thread [5].
- **Multithread application:** There is more than one Thread in the process (context) of that application [5].
- **Non-preemptive:** Once the processor starts its execution, it must finish it before executing the other. It cannot be paused in the middle.
- **Preemptive***:* A processor can be preempted to execute the different processes in the middle of any current process execution.
- **Process (Context):** When a user runs an application, the OS takes an instance of that application into the memory. That instance is called a process (or context of an application). Each process is completely isolated from other processes that run on the same system [5].
- **Process ID (PID):** a unique number that identifies each running process in an operating system. Such as Linux, Unix, Mac OSX, and Microsoft Windows [6].
- **Stack:** Region in memory where local variables are stored and passed into functions [5].
- **Thread:** A thread is also known as a lightweight process and resides inside the process. A thread contains a stack and instruction-pointers [5].

Lecture Notes in Computer Science:
Producer-Consumer Problem in Process Synchronization

1 Introduction:

The basic concept of Multithreading, Concurrency, Performance Optimization, Process scheduler, Context switch, OS Scheduling, and Process Synchronization. Summary of the processes and threads. The producer/consumer problem (in process synchronization) Implementing conditional-variables, semaphores, mutex lock to solve the producer-consumer problem.

2 Process Scheduling and Process Synchronization in Operating System:

2.1 Process:

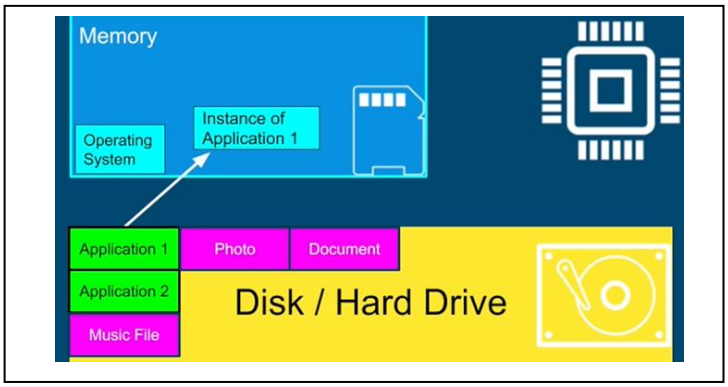


Image retrieved from [5]

In the diagram, Application1 was load from Hard Drive into the Memory by the Operating System. The instance of Application1 is called a process (or context of an application). Each process is completely isolated from other processes that run on a system and has a unique process ID.

2.2 Process Scheduling:

2.2.1 First Come First Served (FCFS):

FCFS is implemented using the Frist In First Out Queue. It is poor performance as the average wait time is high [6]. Its biggest issue is that a *long thread can cause Starvation*. Starvation is worst for User Interface being unresponsive (bad user experience). The UI-thread is usually shorter since its main job is simply getting input from the user and update the screen [5].

2.2.2 Shortest Job First (SJF):

The best approach to minimize waiting time. Easy to implement in Batch systems where required CPU time is known in advance. Impossible to implement in interactive systems where the required CPU time is unknown [6]. The problem is *continuously scheduling* the shortest job first, the longer process may *never be executed* [5].

2.2.3 Priority Scheduling:

This is one of the most common scheduling in the Batch system. Each process is assigned a priority number. A higher priority process will get executed first [6].

2.2.4 Shortest Remaining Time First (SRTF):

SRTF is the preemptive(*) version of Shortest Job First Scheduling. The processor is allocated to the job closest to completion, but it can be preempted by a newer ready job with a shorter time to completion. It has *the same issue as SJF* that it is hard to implement in Interactive System [6].

2.2.5 Round Robin (RR):

In Round Robin, each process is provided a fixed time(quantum) to execute. Once a process is executed for a given period, it is preempted, and another process executes for a given period. Context switching is used to save states of preempted processes [6].

2.2.6 Multiple-Level Queues Scheduling:

Multiple-level queues are a combination of one or more process scheduling algorithm.

2.2.7 Thread Scheduling-Epochs:

Modern OS Linux scheduling algorithm works by dividing the CPU time into *epochs*. In a single epoch, every process has a specified time quantum whose duration is computed when the epoch begins. In general, different processes have different time quantum durations.

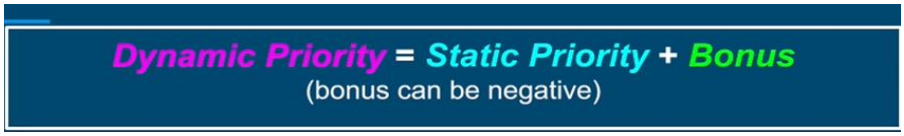


Image retrieved from [5]

- **Static priority:** is set by the developer programmatically.
- **Bonus**: is adjusted by the OS in every epoch for each Thread.
- Using **Dynamic Priority**, the OS will give preference for interactive Threads (such as User Interface Threads). OS will give preference to the thread that *did not complete* in the last epochs or *did not get enough time* to run to prevent Starvation.

2.2.8 Linux 2.6 Completely Fair Scheduler (CFS):

2.2.8.1 Processes vs. threads:

Linux treats *processes and threads the same*. A process can be viewed as a single thread and can also contain multiple threads that share some number of resources (code and/or data) [8].

2.2.8.2 *Overview of CFS:*

The Linux CFS scheduler maintains a time-ordered red-black tree. The red-black tree is an implementation of a self-balancing binary search tree. It mainly maintains the balance between paths in the tree-brands.

Moreover, the operations on the tree occur O(log *n*) time (where *n* is the number of nodes in the tree). This means that system can insert or delete a task quickly and efficiently[8].

2.2.8.3 Operation of the red-black tree:

With tasks (represented by *sched_entity* objects) stored in the time-ordered red-black tree. The highest-needed tasks are stored toward the left side of the tree while the least-needed tasks are stored on the right side. The scheduler picks the left-most node of the red-black tree to schedule next to maintain fairness. The task accounts for its time with the CPU by adding its execution time to the virtual runtime and inserted it back into the tree if runnable. Thus, tasks on the left side of the tree are given time to execute, and the contents of the tree migrate from the right to the left to *maintain fairness*. Therefore, each runnable task chases the other to maintain a balance of execution across the set of runnable tasks [8].

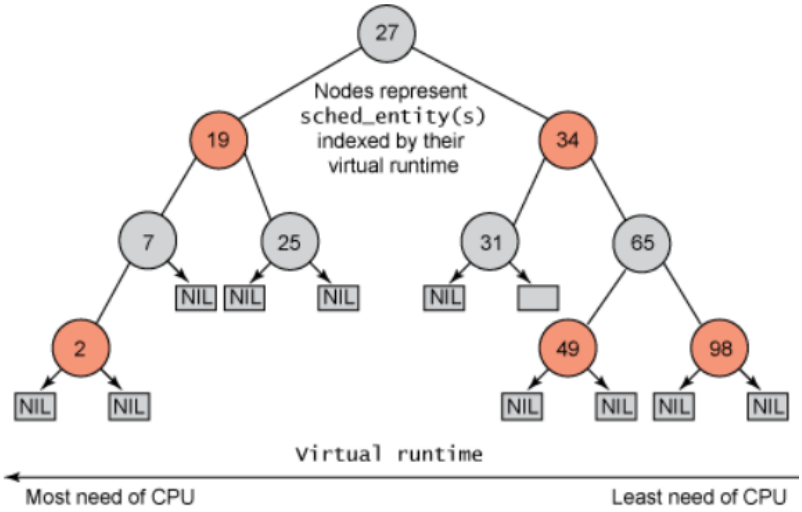


Image retrieved from [8]

2.3 Process Synchronization:

Mainly used for Cooperating Process that shares the resources. It is a technique that is used to coordinate the process that uses shared data. There are two types of processes:

- **Independent Process:**

  The process that isolates (not affected) from the other processes while executing. It does not share any shared resources with other processes.

- **Cooperating Process:**

  The process that affects or is affected by the other process while execution, is called a Cooperating Process. For example, processes that share files, variables, databases, ... with another process.

2.4 Race Condition:

Happen when where several processes try to access the same resources and modify the shared data concurrently. Thus, the outcome of the process depends on the particular order of execution that leads to data inconsistency [7]. This condition can be avoided by using the technique called Process Synchronization which allowing only one process to enter and manipulates the shared data in the Critical Section. This setup can be defined in various sections like:

- **Entry Section:** It is part of the process which decides which process will enter the Critical Section [7].
- **Critical Section:** in critical section, only one process is allowed to enter and modify the shared variable. This part of the process ensures that only no other process can access the resource of shared data [7].
- **Exit Section:** This process allows the other process that is waiting in the Entry Section, to enter into the Critical Sections. It checks that a process that after a process has finished execution in Critical Section can be removed through this Exit Section [7].

- **Remainder Section:** The other parts of the Code other than the Entry Section, Critical Section, and Exit Section are known as the Remainder Section [7].

2.5 Critical Section:

When more than one processes access the same code segment that segment is known as the critical section. The critical section contains shared variables or resources which are needed to be synchronized to maintain consistency of the data variable. In simple terms, a critical section is a group of instructions/statements or regions of code that need to be executed atomically, such as accessing a resource (file, input or output port, global data, etc.).

In concurrent programming, if one thread tries to change the value of shared data at the same time as another thread tries to read the value (i.e. data race across threads), the result is unpredictable.

2.6 Critical Section Problems: Critical Section problems must satisfy these three requirements:

- **Mutual Exclusion:**

  It states that no other process is allowed to execute in the critical section if a process is executing in the critical section.

- **Progress:**

  When no process is in the critical section, then any process from outside that requests for execution can enter the critical section without any delay. Only those processes can enter that have been requested and have a finite time to enter the process.

- **Bounded Waiting:**

  An upper bound must exist on the number of times a process enters so 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.

2.7 Approach:

- **Software Approach**:

Uses an Algorithm approach to maintain synchronization of the data. For example, Peterson's Algorithm uses two variables in the Entry Section to maintain consistency. The two variables are Flag (Boolean) and Turn (to store the process states)[7]. When the condition is True then the process in the waiting State, known as Busy Waiting State.

- **Hardware Approach**:

  Can be done through Lock & Unlock technique. The locking part is done in the Entry Section so that only one process is allowed to enter into the Critical Section, after it completes its execution, the process is moved to the Exit Section, where Unlock Operation is done so that another process in the Lock Section can repeat this process of Execution. This process is designed in such a way that all the three conditions of the Critical Sections are satisfied [7]. This is also explained further below using Conditional Variables to solve the Producer/Consumer Problem (where both processes try to access the same resources).

2.8 Implement the Producer/Consumer Problem in Process Synchronization:

The Producer/Consumer problem is a popular synchronization problem which is proposed by Edsger W. Dijkstra. The Producer/Consumer contains two processes the Producer and the Consumer. The Producer generates and puts data in the Buffer while the Consumer consumes the data (i.e. removing it from the buffer), one piece at a time.

In normal operation, the Consumer Process and Producer Process will not operate at the same time (on the same Buffer). Which means one will sleep (or do other tasks) while the other one operates. In the below implementation, the Buffer is a fixed-size buffer with queue implementation [4].

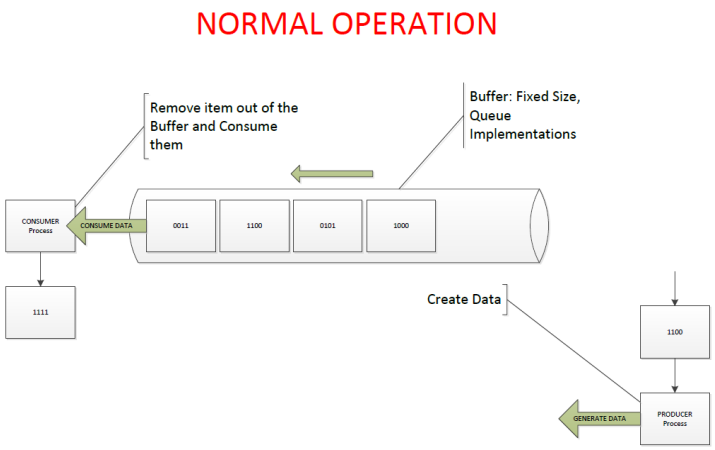


Diagram [A] designed by Truc Huynh

2.8.1 Synchronization problems:

Possible synchronization issues in the Producer-Consumer problem may cause Deadlock or violate the Critical Section (in this case is the Buffer).

2.8.2 Deadlock:

This occurs when a process or thread enters a waiting state because a requested system resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process [4].

In diagram [C], the Consumer Process enters the *waiting-process* (wait for the Producer) when the Buffer is empty. The Procedure Process generates data and fill-up the Buffer. After the Buffer is filled up, the Procedure Process enters the *waiting-process* (wait for the Consumer). The Resource in Buffer is allocated and reserve for the Consumer Process to consume. If the Consumer Process fails to wake up, the system enters a deadlock.

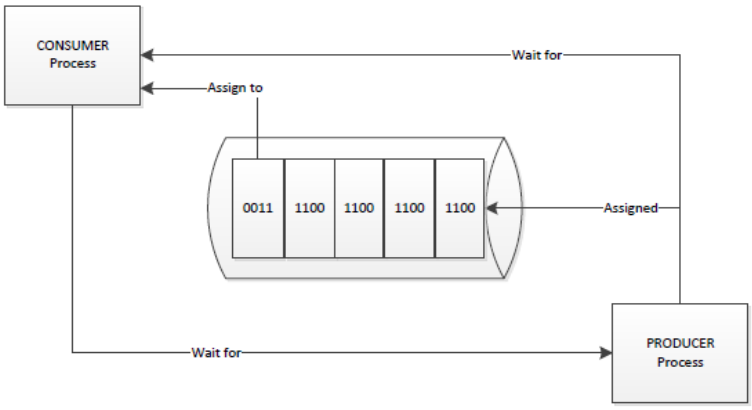


Diagram [C] designed by Truc Huynh

2.8.3 *Consumer Processes and Producer Processes Operate at the same time:*

In diagram [D], the Producer Process and the Consumer Process get access to the Buffer at the same time. By default, one process (either the Consumer or the Producer) must enter the *waiting-process* until the other finished its operation. If the signals fail, the system will enter deadlock when both processes are in the *waiting-process.*

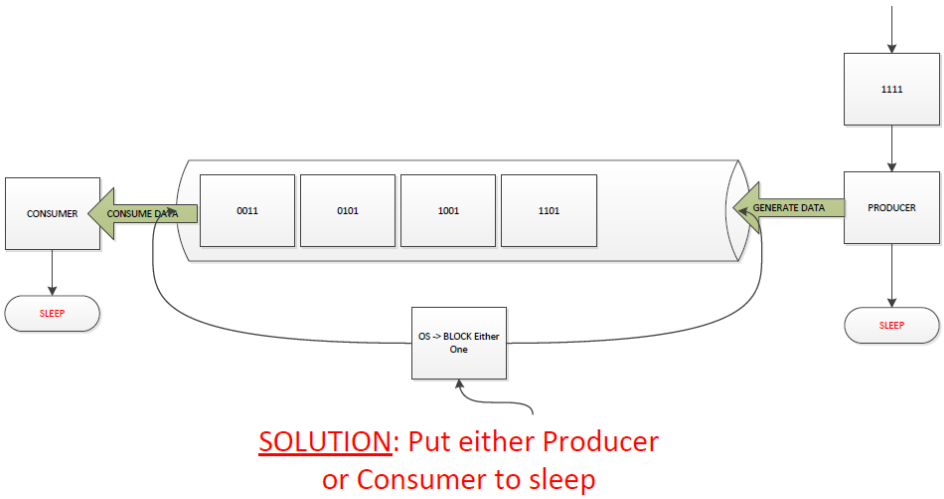


Diagram [D] designed by Truc Huynh

2.8.4 *Consume Empty Buffer:*

In normal operation, the Consumer Process can go to sleep if it finds the Buffer is empty [4]. It will signal the Producer and then enter sleep mode. In diagram [E], the Consumer fails to signal the sleeping Producer (in the *waiting-process*). Since both processes are waiting to be awakened, they enter Deadlock.

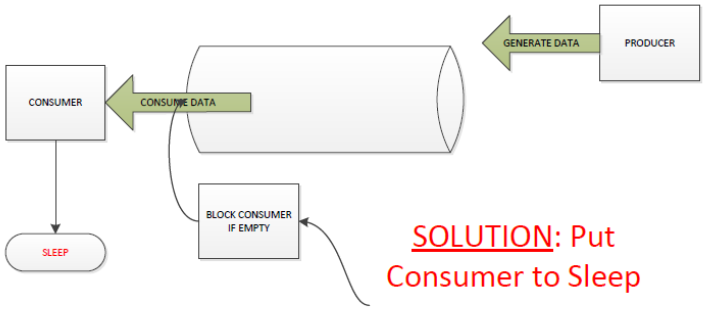


Diagram [E] designed by Truc Huynh

2.8.5 *Add Data to Full Buffer:*

In normal operation, the producer is either go to sleep or discard data if the buffer is full [4]. It will notify the Consumer before entering sleep mode. In diagram [E], the Producer fails to signal the sleeping Consumer (in the *waiting-process*). Since both processes are waiting to be awakened, they enter Deadlock.

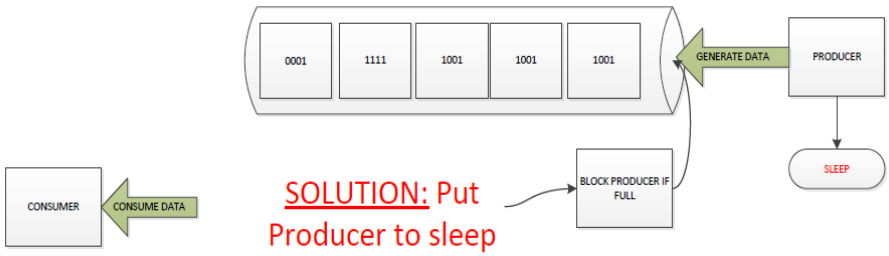


Diagram [F] designed by Truc Huynh

2.8.6 *Solution Implementation:*

Solutions to solve Bound-Buffer problems:

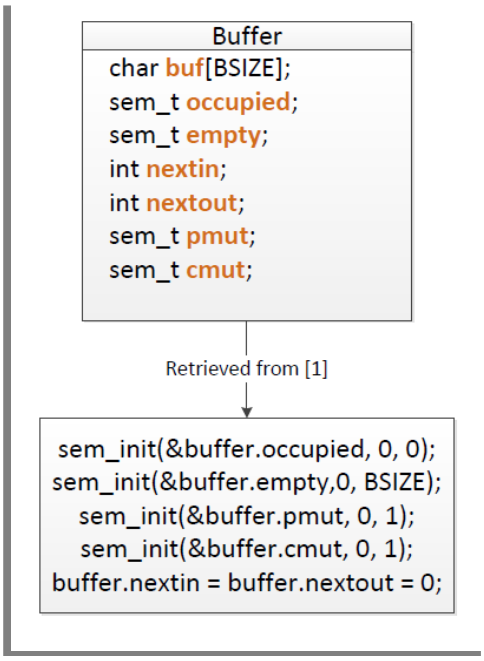
- Mutex Lock
- Conditional Variables

- Semaphores Algorithm
- Peterson Algorithm

2.8.7 Producer and Consumer Problem with Semaphores:

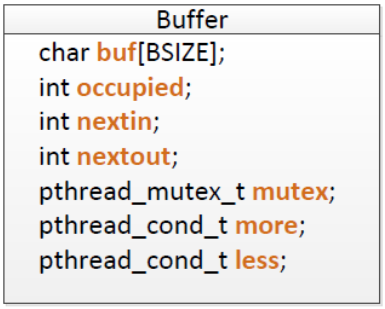
Semaphores were designed by E. W. Dijkstra in late 1960. To use a semaphore, the Thread that wants access to the shared resource tries to acquire a permit [2].

- If the semaphore's count (sem_t) is greater than zero, then the Thread acquires a permit, which causes the semaphore's count to be decremented.
- Otherwise, the thread will be blocked until a permit can be acquired.
- When the Thread no longer needs access to the shared resource, it releases the permit, which causes the semaphore's count to be incremented.
- If there is another thread waiting for a permit, then that Thread will acquire a permit at that time.
- Buffer's data-structures implementation [1]:



2.8.8 Producer and Consumer Problem with Conditional Variables:

2.8.8.1 *Producer/Consumer Problem-The Buffer*



Buffer's data-structures implementation [1]:

2.8.8.2 *Producer/Consumer Problem-The Producer*

The producer thread acquires the mutex protecting the buffer data structure and then makes certain that space is available for the item being produced. If not, it calls **pthread_cond_wait()**, which causes it to join the queue of threads waiting for the condition less, representing there is room in the buffer, to be signaled.

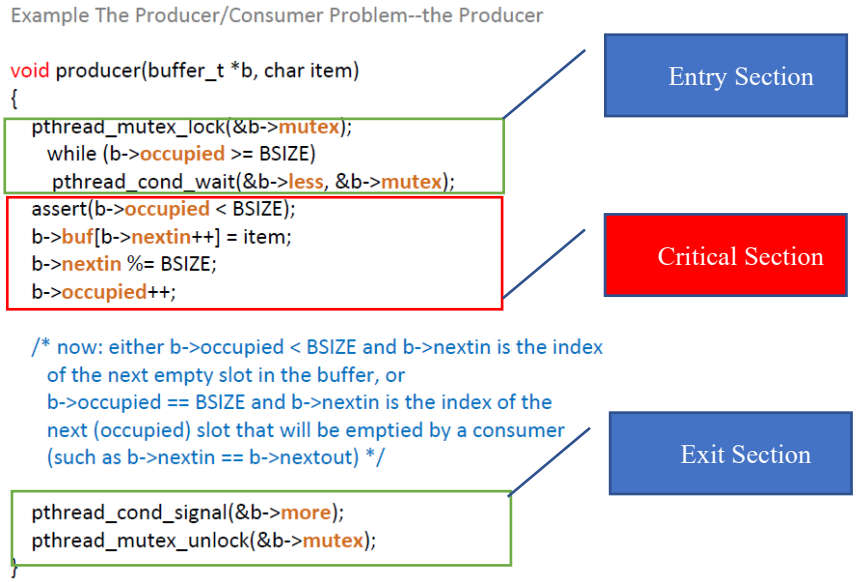
At the same time, as part of the call to **pthread_cond_wait()**, the thread releases its lock on the mutex. The waiting producer Threads depend on consumer Threads to signal when the condition is true (as shown below). When the condition is signaled, the first Thread waiting on less is awakened. However, before the Thread can return from **pthread_cond_wait()**, it must acquire the lock on the mutex again.

This ensures that it again has mutually exclusive access to the buffer data structure. The Thread then must check that there is room available in the buffer; if so, it puts its item into the next available slot.

At the same time, consumer Threads might be waiting for items to appear in the buffer. These Threads are waiting on the condition variable more. A

producer thread, having just deposited something in the buffer, calls **pthread_cond_signal()** to wake up the next waiting consumer. (If there are no waiting consumers, this call does not affect.)

Finally, the producer thread unlocks the mutex, allowing other threads to operate on the buffer data structure.



2.8.8.3 Producer/Consumer Problem-The Consumer:

The assert() statement; unless the code is compiled with NDEBUG defined, assert() does nothing when its argument evaluates to true (nonzero), but causes the program to abort if the argument evaluates to false (zero). Assertions are especially useful in multithreaded programs. they immediately point out runtime problems if they fail, and they have the additional effect of being useful comments.

Both the assertion and the comments are examples of invariants. These are logical statements that should not be falsified by the execution of the program, except during brief moments when a Thread is modifying some of the

program variables mentioned in the invariant. (An assertion, of course, should be true whenever any Thread executes it.)

Using invariants is an extremely useful technique. Even if they are not stated in the program text, think in terms of invariants when you analyze a program.

The invariant in the producer code that is expressed as a comment is always true whenever a thread is in the part of the code where the comment appears. If you move this comment to just after the mutex_unlock(), this does not necessarily remain true. If you move this comment to just after the assert(), this is still true.

The point is that this invariant expresses a property that is true at all times, except when either a producer or a consumer is changing the state of the buffer. While a thread is operating on the buffer (under the protection of a mutex), it might temporarily falsify the invariant. However, once the Thread is finished, the invariant should be true again.

Example The Producer/Consumer Problem--the Consumer

```
char consumer(buffer_t *b)
{
  char item;
  pthread_mutex_lock(&b->mutex);
  while(b->occupied <= 0)
    pthread_cond_wait(&b->more, &b->mutex);
  assert(b->occupied > 0);
  item = b->buf[b->nextout++];
  b->nextout %= BSIZE;
  b->occupied--;
  /* now: either b->occupied > 0 and b->nextout is the index
     of the next occupied slot in the buffer, or
     b->occupied == 0 and b->nextout is the index of the next
     (empty) slot that will be filled by a producer (such as
     b->nextout == b->nextin) */
  pthread_cond_signal(&b->less);
  pthread_mutex_unlock(&b->mutex);
  return(item);
}
```

Entry Section

Critical Section

Exit Section

3 The role of Thread in Operating System:

3.1 Responsiveness:

Responsiveness is the ability to allow multiple users simultaneously (ex: online store) access to the resources without interrupting [5]. It is particularly critical in applications with user interfaces where instance-feedback is provided from the screen. Responsiveness can be achieved by using multiple threads with separate threads for each stack [5].

Concurrency (multitasking) can achieve by multitasking between threads. For example, computers create the illusion that all the tasks happening at the same time. We do not need multiple cores to achieve concurrency [5].

3.2 Performance (Thread):

Threads can increase the system's performance by parallelism. With multiple cores, we can truly run tasks completely in parallel. The benefit is to complete complex tasks much faster and finish more work in the same period. For high-scale service, fewer machines are used which means less money is spent on infrastructure.

3.3 Multithreading Caveat:

Multithread programming is fundamentally different from single Thread programming. Fundamentals of single-thread programming may break if they are applied to multithread programming.

3.4 Single Thread application:

In a single thread application, the process contains metadata such as process ID (PID), files (that application open for reading and writing), code, data (heap), main Thread (stack and instruction pointer). Please refer to Keyword-section for definition.

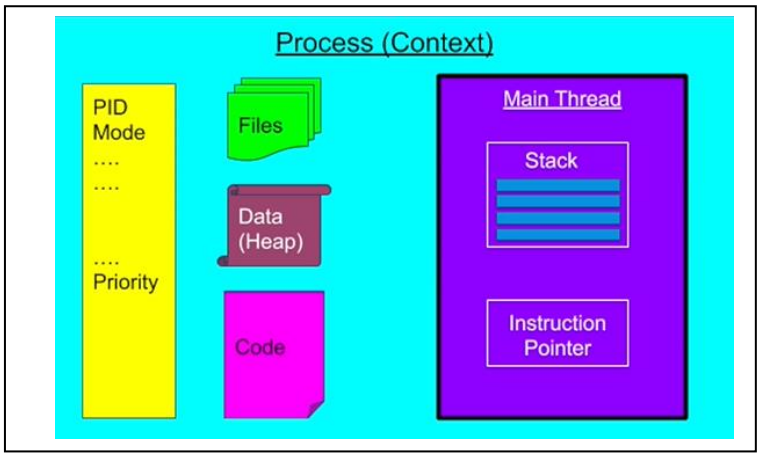


Image retrieved from [5]

3.5 Multithread application:

In a multithread application, there is more than one thread in the process (context). All the Threads in the same process of multithread application will share the same resources (Process ID (PID), Files, Code, Data (Heap)…). Please refer to Keyword for definition.

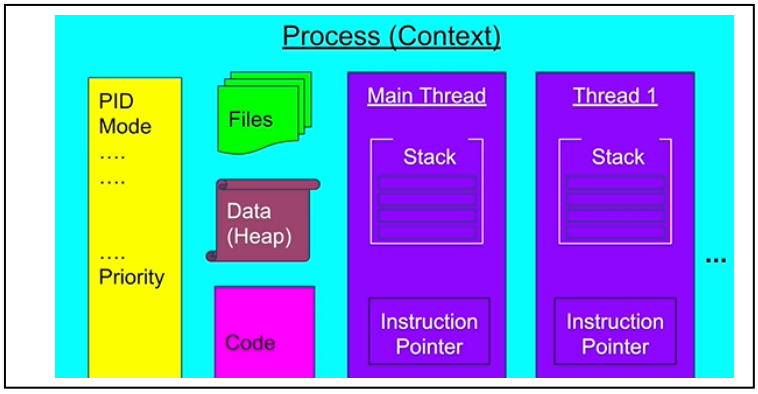


Image retrieved from [5]

3.6 Context Switch:

Each instance(process) has a unique process ID and may have one or more Threads. All the threads are competing with each other to be executed in the CPU. However, there is always more Thread than Core, so the OS has to run 1 Thread and stop it then run another

Thread. The act of stopping a running thread and start a new one is called a context switch[5].

Context Switch Cost [5]:

- Context switch cost is not cheap and is the price of multitasking (Concurrency)
- Same as humans when we multitask. Take time to focus.
- Each Thread consumes resources in the CPU and memory.
- When we switch to a different Thread, we need to store the data for the current Thread and restore data for another Thread

3.7 Thread Scheduling:

Thread Scheduling is the way the OS decides when to run which Thread and when to perform a context switch. There is no guarantee that which runnable Thread will be chosen to run by the thread scheduler. Only one thread at a time can run in a single process. The Thread scheduler mainly uses preemptive or time-slicing scheduling to schedule the Threads [5].

3.8 Thread vs. Process:

When to prefer multithreading architecture:

- When the tasks share a lot of resources since threads are much faster to create and destroy.
- Switching between threads of the same process is much faster (shorter context switches)

When to prefer multi-process architecture:

- Security and stability are of higher importance than performance.
- Tasks are unrelated to each other.

3.9 Conclusion:

- If there are too many running-threads, the Operating System will spend more time in management Threads (perform the context switch) than real productive work. This is called Thrashing.
- Thread consumes fewer resources than process.
- Context Switch between two Threads from the same process is cheaper (cost fewer resources) than context switch between different processes.

4 Java Thread:

Java-threads are objects (called virtual threads). The thread object is managed by the JVM. Therefore, their allocation does not require a system call, and they are free of the operating system's context switch. Furthermore, thread objects run on the carrier thread, which is the actual kernel thread used under-the-hood. Each thread object is associated with an instance of class Thread (java.lang.Thread). There are two basic strategies for using thread-objects in Java to create a concurrent application.

- To directly control Thread creation and management, simply instantiate Thread each time the application needs to initiate an asynchronous task.
- To abstract Thread management from the rest of your application, pass the application's tasks to an executor.

4.1 Direct Control:

The Java Virtual Machine allows an application to have multiple threads of execution running concurrently. Direct control uses for small applications. The developer simply defined a new Thread as defined by its Runnable object, and the thread itself, as defined by a Thread object.

4.2 Abstract Thread Management:

Extremely useful for large-scale applications, the most approach is to separate thread management and creation from the rest of the application. The object that encapsulates these functions are known as the executor.

4.3 Executor Interfaces:

Retrieved from [8], the java.util.concurrent package defines three executor interfaces:

- Executor, a simple interface that supports launching new tasks.
- Executor-Service, a sub-interface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
- Scheduled-Executor-Service, a sub-interface of Executor-Service, supports future and/or periodic execution of tasks.

4.4 Thread Pools:

- Most of the executor implementations in java. util.concurrent use thread pools, which consist of worker threads. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks [9].
- Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead [9].
- An important advantage of the fixed thread pool is that applications using it *degrade gracefully*. For example, a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to *all* requests when the overhead of all those threads exceeds the capacity of the system. With a limit on the number of threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain [9].

- A simple way to create an executor that uses a fixed thread pool is to invoke the *newFixedThreadPool* factory method in *java.util.concurrent.Executors* [9].

5. Conclusion:

Except for the summary of definition (threads and process). The report explains the advantage of Time Scheduling Epoch (Dynamic Priority) & Linux Completely Fair Scheduling vs. traditional scheduling, multithreading vs. single thread vs. multi-process, Java Virtual Thread, Abstract Thread Management, Fixed Thread Pool, Synchronization vs. Concurrency. The report also using the Producer/Consumer problem to demonstrate the use of the critical section (synchronization) and using Conditional Variables to ensure synchronization.

References

Jones, M.,(Sept, 2018) *Inside the Linux 2.6 Completely Fair Scheduler-IBM Developer*. Retrieved from Inside the Linux 2.6 Completely Fair Scheduler – IBM Developer **[8]**

Oracle(n.d.) *The Producer/Consumer Problem*. Retrieved from https://docs.oracle.com/cd/E19455-01/806-5257/sync-31/index.html **[1]**

Oracle (n.d.) Executor Interfaces. Retrieved from Executor Interfaces (The Java™ Tutorials > Essential Classes > Concurrency) (oracle.com)[9]

Miglani., G.(2018) *Semaphore in Java.* Retrieved from https://www.geeksforgeeks.org /semaphore-in-java/ **[2]**

Pogrebinsky., M., (2020) Java Multithreading, Concurrency & Performance Optimization. Retrieved from Java Multithreading, Concurrency & Performance Optimization | Udemy [5]

Shah., K.; Rithesh (2020) *Introduction of Deadlock in Operating System*. Retrieved from: https://www.geeksforgeeks.org/ introduction-of-deadlock-in-operating-system/ **[3]**

Shuka., K.,(2019) *Process Synchronization*. Retrieved from Process Synchronization | Set 2 - GeeksforGeeks **[7]**

Tutorial Point (n.d.) *Operating System Scheduling algorithms.* Retrieved from Operating System Scheduling algorithms - Tutorialspoint **[6]**

Yadav, G.,(2019) *Producer-Consumer solution using threads in Java.* Retrieved from https://www.geeksforgeeks.org/producer-consumer-solution-using-threads-java/ **[4]**