# Operating System

**Deadlocks**

Deadlock will occur if all the below conditions are held true.

1. Mutual exclusion: only one process can have a resource at a time.
2. Hold and wait: a process is waiting for another resource while it has already held a previous resource.
3. No preemption: a resource can be release only voluntarily by the process holding it.
4. Circular wait: there exists a circular wait such that P1 waiting for a resource acquired by P2, P2 waiting for a resource acquired by P3, P3 for P4, and so on, until P(n) for P1.

How to prevent a deadlock?

1. Ensure circular wait doesn’t occur, i.e., maintaining an order of resource acquisitions (lock ordering/locking sequence/keeping hierarchy of locks).
2. Rest other conditions mentioned above are desirable (1-3), so we cannot compromise with them.

**Banker’s algorithm**

Deadlock avoidance algorithm to check if a process’ request for resources should be fulfilled or not, and whether it can cause a deadlock or not.

**Program layout:**

BSS or data segment, both are said to be allocated at compile time as they get addresses where they are going to be stored. Each process running the same program has its own BSS area. While running, the BSS data is placed in the data segment. In the executable file, they are stored in the BSS section.

**BSS:**

What type of variables go to BSS?

1. Global uninitialized or 0 initialized.
2. Global static uninitialized or 0 initialized.
3. Local static POD types uninitialized or 0 initialized.
4. Local static objects (non-POD) types. They are initialized at the time when function is called, hence before that, they’ll be in BSS no matter they are initialized or not. They cannot be in stack as their lifetime is program life time but stacks are deallocated once the function call returns.
5. Static uninitialized POD in class.
6. Static object member in class no matter they are initialized or not.

Keeping a variable in BSS means it is given an address, but it is not initialized.

1. **Global variable** and **static member in class**: constructor is called before entering **main** function. For static member in class, it needs to have initialization.
2. **Local static variable**: constructor is only called when execution reaches its declaration for the first time.
3. If **Local static variable is POD type**, then it is also initialized before enter **main** function.

Example for POD type: *static int number = 10;*

<https://stackoverflow.com/questions/55510/when-do-function-level-static-variables-get-allocated-initialized>

Static variable memory allocation

The compiler assigns a static memory address to the array:

C++ code:

int a[4];

int b[] = { 1 , 2 , 3 , 4 };

int main()

{

}

Output assembly:

a:

.zero 16

b:

.long 1

.long 2

.long 3

.long 4

main:

pushq %rbp

movq %rsp, %rbp

movl $0, %eax

popq

**static and dynamic libraries**

* exit() and \_exit() : The following actions are performed by exit() :

exit handlers (functions registered with atexit() and on\_exit() ) are called, in

reverse order of their registration. The stdio stream buffers are flushed. The \_exit() system call is invoked.

* return 0 and exit(0) : *return 0* returns from the main function to the caller function (\_start\_) and then calls *exit(0)*. If any steps performed during exit processing access variables local to main(), then doing a return from main() results in undefined behavior.

In C++, local static objects aren’t destroyed if *exit* is called whereas they are in case of *return*. static objects are destroyed however.

**Size of a file C program**

FILE\* fp = fopen(file\_name, "r");

fseek(fp, 0L, SEEK\_END);

long int res = ftell(fp);

**Interrupts**

Process context:

Interrupt context: When executing an interrupt handler, the kernel is in interrupt context. It should be fast and should not sleep as it has interrupted other process asynchronously. Also, it cannot call any function that invokes sleep, thereby limiting the number of functions it can call.

Why can’t you sleep in an interrupt handler?

Kernel doesn’t run all the time. In a single core system, when a user mode process is running, kernel would start running when the process invokes a system call. Kernel can be scheduled as a result of the interrupt generated on system call. Similar for hardware interrupts. Operating systems run through exceptions and interrupts. If there are no system calls, and no hardware interrupts, the kernel does nothing because it is not entered - there is nothing for it to do. A single process would keep on running in this case because there is no timer expiry interrupt (or no system calls made) to make use of the scheduler (or system call handlers) which is part of the kernel.

**32-bit and 64-bit processors**

This basically means the size of the registers in CPU. Registers may hold addresses that are written onto address bus, hence address bus is also the size of registers. Data bus transfers information from one hardware to another, and it has lines for that. The number of lines in the data bus determine the word size (the basic unit of data CPU works upon). If a data bus has 64 lines then word size is 8 bytes.

Therefore, we can say that a 64-bit processor will address 2^64 different addresses. It makes sense to have a 32-bit CPU with a 32-bit data bus because you can transfer all data from memory directly to the registers, but you can have any data bus size. So, a 32-bit CPU normally have 32-bit data bus to make it easy to transfer data from and to it.

Volatile memory is [computer memory](https://en.wikipedia.org/wiki/Computer_memory) that requires power to maintain the stored information; it retains its contents while powered on but when the power is interrupted, the stored data is quickly lost.

Difference between thread and process scheduling?

TLB flushes might not occur on thread context switch.

**Program layout:**

BSS or data segment, both are said to be allocated at compile time as they get addresses where they are going to be stored. Each process running the same program has its own BSS area. When running, the BSS data are placed in the data segment. In the executable file, they are stored in the BSS section.

*BSS*:

What type of variables go to BSS?

1. Global uninitialized or 0 initialized.
2. Global static uninitialized or 0 initialized.
3. Local static POD types uninitialized or 0 initialized.
4. Local static objects (non-POD) types. They are initialized at the time when function is called, hence before that, they’ll be in BSS no matter they are initialized or not.
5. Static object member in class no matter they are initialized or not.
6. Static POD in class uninitialized.

Keeping a variable in BSS means it is given an address, but it is not initialized.

1. Consider the following scenarios:

int f(){

int a = 5;

static int b = a; // stored in bss section

}

int f(){

int a = 5;

static int b = a; // stored in bss section

}

int f(){

const int a = 5;

static int b = a; // stored in data section with value 5

// however no memory allocated to ‘a’

}

1. **Global variable** and **static member in class**: constructor is called before enter **main** function.
2. **Local static variable**: constructor is only called when execution reaches its declaration at first time.
3. If **Local static variable is POD type**, then it is also initialized before enter **main** function.

Example for POD type: *static int number = 10;*

<https://stackoverflow.com/questions/55510/when-do-function-level-static-variables-get-allocated-initialized>

**.h and .c files**

* .h files have declarations and .c have definitions.
* We only include .h files as compiler only needs declarations.
* It’s linkers responsibility to find the definitions in the compiled library’s object file(s) present at standard path by searching through the files. We can also define our own path for dynamic linked libraries.
* In static linked binaries, the definition of the functions will be appended in the program binary itself, whereas in dynamic linked binaries it contains only references.
* exit() and \_exit() : The following actions are performed by exit() :

exit handlers (functions registered with atexit() and on\_exit() ) are called, in

reverse order of their registration. The stdio stream buffers are flushed. The \_exit() system call is invoked.

* return 0 and exit(0) : *return 0* returns from the main function to the caller function (\_start\_) and then calls *exit(0)*. If any steps performed during exit processing access variables local to main(), then doing a return from main() results in undefined behavior.

File pointer contains File descriptor and has buffering mechanism. Doesn’t have portability issues for programs.

**Interrupts**

Process context:

Interrupt context: When executing an interrupt handler, the kernel is in interrupt context. It should be fast and should not sleep as it has interrupted other process asynchronously. Also, it cannot call any function that invokes sleep, thereby limiting the number of functions it can call.

Why can’t you sleep in an interrupt handler?

In a clocked design, a complex circuit is broken down into progressively smaller blocks by adding latches so that the input and output of each block changes only when triggered by a clock pulse.

In an asynchronous design, the amount of time it will take a given block to transform its input into a valid output is somewhat *unpredictable*. This means every block needs a mechanism to signal that its output has become valid, and a mechanism to check for when its inputs become valid.

Hardware interrupts are asynchronous. They can happen in the middle of an instruction execution, i.e., they occur asynchronously with respect to the processor clock. This is because clock of other hardware is not in sync with the clock of the processor. If multiple instructions are executed per clock cycle, then we can say that a hardware interrupt is asynchronous with CPU’s clock.

Interrupts generated by processor (divide by zero) are synchronous interrupts. They will occur after the current clock cycle.

System calls are software interrupts. Synchronous means they are synchronized with processor clock.

Both latter two are called *exceptions*.

Kernel doesn’t run all the time. In a single core system, when a user mode process is running, kernel would start running when the process invokes a system call. Kernel can be scheduled as a result of the interrupt generated on system call. Similar for hardware interrupts. Operating systems run through exceptions and interrupts. If there are no system calls, and no hardware interrupts, the kernel does nothing because it is not entered - there is nothing for it to do. A single process would keep on running in this case because there is no timer expiry interrupt (or no system calls made) to make use of the scheduler (or system call handlers) which is part of the kernel.

**Compiling a C program**

1. Preprocessing is the first pass of any C compilation. It processes include-files, conditional compilation instructions and macros.
2. Compilation is the second pass. It takes the output of the preprocessor, and the source code, and generates assembler source code.
3. Assembler takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an object file.
4. Linking is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single executable file. It resolves references to external symbols (like extern variables), assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called relocation).

.text

contains the executable instruction codes and is shared among every process running the same binary.

.bss

It holds un-initialized global and static variables. The size that BSS will require at runtime is recorded in the object file, but the BSS (unlike the data section) doesn't take up any actual space in the object file.

.data

Contains the initialized global and static variables and their values. It is usually the largest part of the executable. It usually has READ/WRITE permissions.

.rdata

.rodata (read-only data) section. This contains constants and string literals.

.reloc

Stores the information required for relocating the image while loading.

Symbol table

Relocation records

Buffer overflow:

When a [program](https://en.wikipedia.org/wiki/Computer_program), while writing [data](https://en.wikipedia.org/wiki/Data_(computing)) to a [buffer](https://en.wikipedia.org/wiki/Buffer_(computer_science)), overruns the buffer's boundary and overwrites adjacent [memory](https://en.wikipedia.org/wiki/Main_memory) locations.

**OS primitives regarding locking and sleeping**

1. How is atomicity implemented? How are atomic operations carried out?
   1. Single processor:
      1. Disabling interrupts which performing operations that have multiple instructions because context switch can compromise atomicity.
      2. Using Spinlocks
   2. Multiprocessor:
      1. You must also ensure that no other core in the system attempts to access the data you are working with. The easiest way to achieve this is to assert the 'LOCK' signal on the bus (bus connecting RAM and processor), which prevents any other processor in the system from accessing the memory at the same time.
2. How is spin lock implemented?
   1. Spinlocks use atomic operations like ‘test and set’ for their implementation. These atomic operations are facilitated by CPU architecture.

// \_\_sync\_bool\_compare\_and\_swap is an atomic instruction that simply checks if the value behind the given pointer equals the second parameter, if so replaces it with the third parameter and returns true - otherwise it does nothing, returns false and our lock keeps spinning:

void acquire\_mutex(mutex\_t\* mutex)

{

while(!\_\_sync\_bool\_compare\_and\_swap(&mutex, 0, 1)) // Atomic instruction

{

asm("pause");

}

}

void release\_mutex(mutex\_t\* mutex)

{

\*mutex = 0;

}

1. How is busy wait implemented?
2. wake\_up\_process() implementation (explicit wakeup even when timeout isn’t over).

Semaphore signaling can lead to “waking up a process”. This is an attempt to wake up the nominated process and move it to the set of runnable processes.

The waiting process can be in waiting state.

p->state = TASK\_RUNNING;

1. How is sleep implemented?
2. Applications:
   1. Producer consumer
   2. Peterson
   3. Dining philosophers

**Semaphores**

1. wait() decrements the value of the semaphore, and if the semaphore is negative, puts the process on the waiting queue until the semaphore is released by the process holding it.
2. signal() increments the semaphore and, if it is still negative, indicates to the scheduler to wake the next waiting process in the queue.
3. Negative value is can be represented as the number of waiting processes in

sem->wait\_list.

* *Use semaphores when designing any data structure.*

struct semaphore {  
        raw\_spinlock\_t          lock;  
        unsigned int            count;  
        struct list\_head        wait\_list;  
};

* lock :- lock which needs to be acquired before doing any operation on semaphore (i.e. updating variable count)
* count :- if count > 0 (e.g count = 5) => semaphore acquired, otherwise current process will wait along with other processes in the 'wait\_list' and may sleep.
* **down() and up() function** (these are deprecated, interruptible versions are to be referred)

int down\_interruptible(struct semaphore \*sem)  
{

// here if count > 0 then decrease it and semaphore is acquired successfully

       raw\_spin\_lock\_irqsave(&sem->lock, flags);

        if (likely(sem->count > 0))  
                sem->count--;  
        else {  
        // if no more tasks are allowed to acquire the semaphore

// i.e., count is already equal to 0, then this process will sleep  
                result = \_\_down\_interruptible(sem);  
         }

raw\_spin\_unlock\_irqrestore(&sem->lock, flags);  
}

struct semaphore\_waiter {

struct list\_head list;

struct task\_struct \*task;

int up;

};

/\* finally \_\_down\_interruptible() will boils down to \_down\_common() \*/

static inline int \_\_sched \_\_down\_common(struct semaphore \*sem, long state,

long timeout)

{

struct task\_struct \*task = current;

struct semaphore\_waiter waiter;

list\_add\_tail(&waiter.list, &sem->wait\_list);

waiter.task = task;

waiter.up = 0;

for (;;) {

// Let say sleep in got interrupted

if (signal\_pending\_state(state, current))

goto interrupted;

if (timeout <= 0)

goto timed\_out;

raw\_spin\_unlock\_irq(&sem->lock);

timeout = schedule\_timeout(timeout);

// the current task sleep until @timeout jiffies have elapsed.

raw\_spin\_lock\_irq(&sem->lock);

if (waiter.up)

return 0;

}

timed\_out:

list\_del(&waiter.list);

return -ETIME;

interrupted:

list\_del(&waiter.list);

return -EINTR;

}

void up(struct semaphore \*sem)

{

raw\_spin\_lock\_irqsave(&sem->lock, flags);

// Since no other process is waiting on the semaphore, it will be incremented by 1

if (likely(list\_empty(&sem->wait\_list)))

sem->count++;

else

\_\_up(sem);

raw\_spin\_unlock\_irqrestore(&sem->lock, flags);

// Hey some processes may be waiting so needs to

// wake one of them. In this case, we are not incrementing sem->count

}

// Takes an entry from sem->wait\_list and sets its up flag to 1

static noinline void \_\_sched \_\_up(struct semaphore \*sem)

{

struct semaphore\_waiter \*waiter = list\_first\_entry(&sem->wait\_list,

struct semaphore\_waiter, list);

list\_del(&waiter->list);

waiter->up = 1;

wake\_up\_process(waiter->task); // p->state = TASK\_RUNNING;

}

<https://www.quora.com/How-are-mutexes-and-semaphores-different-with-respect-to-their-implementation-in-a-Linux-kernel>

**Mutex Locks**

* These have similar implementation like that of semaphore (for interview answer).
* In case of semaphore, when a semaphore-releasing process sees waiting processes in the sem->wait\_list, they would set their waiter->up flag to 1, implying in essence, that one process signaled another process.
* In case of mutex, when one process takes out another waiting process from the wait\_list, then that woken up process again tries the below operation:

MUTEX\_SHOW\_NO\_WAITER(lock) &&  
               (atomic\_xchg(&lock->count, -1) == 1))  
                goto done;

**Recursive Mutexes**

A thread can acquire the same lock multiple times

1. When it could make sense:
   1. F1 and F2 both need to be synchronized. They can be called one after the other or simply independent.
2. When it could not make sense:
   1. Used in a recursive function call.

**Producer Consumer Problem**

* This problem can be solved by semaphores when there is only 1 Producer and 1 Consumer.

We keep a count of filledCount and emptyCount. We are keeping two semaphores instead of one because we want the consumer to wait if fillCount is 0. Suppose we had only used fillCount and fillCount was equal to BUFSIZE, then we also need to stop the producer from adding more elements, hence we keep emptyCount so that the producer can wait until emptyCount is 0.

semaphore fillCount = 0; // items produced

semaphore emptyCount = BUFFER\_SIZE; // remaining space

procedure producer()

{

while (true)

{

item = produceItem();

down(emptyCount);

putItemIntoBuffer(item);

// consumer doesn’t know yet that an item has been put into the buffer as it only

// checks fillCount

up(fillCount);

}

}

procedure consumer()

{

while (true)

{

down(fillCount);

item = removeItemFromBuffer();

// producer doesn’t know yet that an item has been popped from the buffer as it only

// checks emptyCount

up(emptyCount);

consumeItem(item);

}

}

* When there are multiple producers/consumers writing to the same buffer, we need an extra mutex to synchronize the offsets in the buffer to which each Producer may write.

procedure producer()

{

while (true)

{

item = produceItem();

down(emptyCount);

down(buffer\_mutex);

putItemIntoBuffer(item);

up(buffer\_mutex);

up(fillCount);

}

}

* The P-C problem can be done without semaphores in case of 1 producer and consumer.

The idea is to first put the element into the buffer and then increment filledCount. Also when there is no element in the buffer, the consumer doesn’t go to sleep; it keeps polling.

*The idea of semaphores is to not do busy wait.*

* Below, even if the producer had pushed an item into the buffer and incremented the filledCount to 1 from 0, it may happen that the consumer might not have seen that change. But the consumer will eventually see that in the next loop.

Similarly, if a producer finds that the emptyCount is equal to the 0 even if an item was popped from the buffer, the producer will eventually get the updated size (increased by some amount) in the next loop.

* The assumption we have made here is that even though the changes in filledCount/emptyCount are not instantaneously visible but they will be visible when read *correctly* in the next cycle. The two Count variables need to be small enough that the processor performs reads and writes of the variable atomically. Otherwise, there is a race condition where the other thread reads a partially-updated and thus wrong value.

volatile unsigned int produceCount = 0, consumeCount = 0;

TokenType sharedBuffer[BUFFER\_SIZE];

void producer(void) {

while (1) {

while (produceCount - consumeCount == BUFFER\_SIZE) {

schedulerYield(); /\* sharedBuffer is full \*/

}

/\* Write to sharedBuffer \_before\_ incrementing produceCount \*/

sharedBuffer[produceCount % BUFFER\_SIZE] = produceToken();

++produceCount;

}

}

void consumer(void) {

while (1) {

while (produceCount - consumeCount == 0) {

schedulerYield(); /\* sharedBuffer is empty \*/

}

consumeToken(&sharedBuffer[consumeCount % BUFFER\_SIZE]);

++consumeCount;

}

}