Operating Systems: Shortnotes

Hardik Rajpal

November 8, 2023

Contents

Chapter 1

Introduction

Chapter 2

CPU Virtualization

Terms

- Process Control Block: A per-process data structure that tracks process meta-data.

Chapter 3

System Calls

3.1 List Of SysCalls (C)

The full list is available here.

3.1.1 Generic

1. `exit(int status)`
    - terminate the calling process.

3.1.2 Files

1. `int dup(int oldfd)`
    - Duplicates a file descriptor.
    - Returns the new file descriptor. (All dup calls return the new file descriptor.)
2. `int dup2(int oldfd,int useThisFd)`
    - Duplicates oldfd, and assigns copy to useThisFd.
    - Silently closes the file on useThisFd if it is already in use.
    - Returns useThisFd.

3.1.3 Processes

1. `excve()`
    - Loads a new program into the address space of the running process.
2. `int fork()`
    - creates a child process with same code and PC value; returns pid in parent process, 0 in child process.
3. `waitpid(pid_t pid, int *stat_loc, int options)`
    - Wait for a child process to stop.
    - Use pid arg as NULL to get status for any child whose group ID matches that of the calling process's group ID.
    - Return value is the pid of the child whose status is reported.

4. `getpid()`
    - Returns PID of calling process.
5. `getppid()`
    - Returns the PID of the calling process's parent process.
6. `int pipe(int pipefd[2], int flags)`
    - Creates a unidirectional data channel that can be used for interprocess communication.
    - pipefd[0] is for reading, and pipefd[1] is for writing.
    - Data written to the pipe is buffered by the kernel until it is read.
    - Returns 0 on a successfuly setup.

Chapter 4

Memory Virtualization

Terms

1. Address Space: The running program's view of the memory available to it. Every program only sees the memory available to it, which is split between the code, (static) data, heap and stack.
2. Sparse Address Space: An address space where most of the memory between heap and stack is unused.
3. Memory Management Unit: The hardware responsible for V2P translation.
4. Protection Bits: Bits in VA that represent the process' access permission for the PA to which it points, including read/write/execute permissions.
5. Fragmentation: Wastage of physical memory space.
    - Internal: Wastage within a contiguous block of physical memory allocated to a process. An example is the unused space between heap and stack in a simple base-bounds V2P map.
    - External: Wastage outside of contiguous blocks that have been allocated. This is because of the gaps of available physical memory being too small to fit any new contiguous segments.
6. Page: Unit of address space.
7. Page Frame: Unit of physical address space to which a page can be mapped. Same unit, but **pages** exist in the virtual address space whereas **page frames** exist in the physical address space.
8. Spatial Locality: nearby instructions (in the control flow) access nearby memory locations.
   Ex. with code for sequential array processing.
9. Temproal Locality: nearby instructions (in the control flow) accessing a single memory address repeatedly.

4.1 Address Space

The program is actually loaded into random physical addresses, but due to the address space abstraction, the program sees the memory available to it as a contiguous chunk, starting at 0. Every address the program sees is virtual and the OS (with some translation mechanism of the VA to PA) uses the PA whenever the program references the VA.

4.2 Address Translation

All address translation is hardware-based for efficiency. Each memory access (load/store) is intercepted by the hardware and the VA is translated to a PA. The OS helps setup the hardware for the right translation (per process), and manages memory.

4.2.1 Dynamic Relocation a.k.a. (Base, Bound)

- Two registers for this in the CPU: base and bound.
- Translation:

```
V2P(VA v){
    if(v > bound || v < 0){
        throw "Out of Bounds";
    }
    return (base + v);
}
```

- Values of base and bounds for each process are stored in its PCB.
- Base and bound registers are privileged: if access is attempted from user mode, the OS terminates the access-requesting process.

4.2.2 Segmentation

- Generalized base and bounds for each logical segment of each process: code, heap and stack.
- Maintain base and size for each segment:

Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

- Use VA[0:2] to represent the segment type.
- Better handles **sparse address spaces**, where the program often has very little heap and stack data and thus a lot of the in-between space in a contiguous allocation is wasted.
- Translation is more involved:

```
V2P(VA v){
    segment = v[0:2];//or bit operations.
    offset = v[2:];//or bit operations.
    if(offset < 0 || offset > segdata[segment]["bound"]){
        throw "Out of Bounds"
    }
    else{
        return segdata[segment]["base"] + offset;
    }
}
```

- OS responsibilities:
  - On each context switch the **segmentation table** for the process is replaced by the incoming process' table.
  - On a receiving new process (and its accompanying address space), the OS has to find space in the physical memory for its segments. If the segments are of varying sizes (bounds) this is more involved.
  - Variable size segments lead to external fragmentation.
  - A solution to external fragmentation is periodic compaction:
    1. Stop running processes.
    2. Copy their data to a contiguous chunk of memory.
    3. Update their segment register values.

This creates available contiguous chunks of physical memory but compaction is expensive (copying segments is memory-intensive) and would take up a fair amount of CPU time.

  - Alternative approaches involve using a free-list management algorithm like:
    * best-fit
    * worst-fit
    * first-fit
    * buddy algorithm
  - External fragmentation will always exist in this scheme, however. The algorithms above simply aim to minimize it. The real solution is to disallow variable sized segments.

- Some systems merge code and heap segments to use only one bit to represent segment.
- There are **implicit** approaches to identify the segment too, where the program infers the segment by identifying how the VA was conceived:
  - from a PC $\implies$ use code segment.
  - from ebp $\implies$ use stack segment.
  - else, use heap segment.
- We also need an additional bit to identify the direction of growth of the segment, if this varies across the segments. The translation function is modified to take this into account.

Sharing Segments

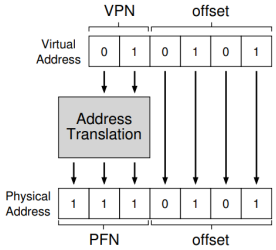
- With some VA bits used to represent permissions that a process has for the segment **protection bits**, we can factor include segments with varying permissions.
- Translation function is appropriately modified to raise an exception whenever a process attempts to violate permissions for a PA.
- Read-only code can be shared across multiple processes, without the worry of harming isolation.

Fine-grained Vs. Coarse-grained

- Segments of code, heap and stack are relatively large (coarse-grained segmentation).
- Fine-grained segmentation further splits up each of these logical segments and identifies each segment using a **segment table**. This allows for better memory management by the OS (by moving unused segments to disk).

4.2.3 Paging

- Split the address space (abstraction visible to the process) into fixed-size units called pages. The corresponding contiguous units in the physical address space are called page frames.
- To record each page's mapping for each process, the OS maintains a **page table** per process that is pointed to by the **Page Table Base Register (PTBR)**.
- Translation: (neglecting bits other than VPN and offset)



Inside The Page Table

- The OS stores the page table in memory, in units of **page table entries** (PTEs) because of the sheer size of page tables.
- Each PTE has a physical address to which it maps and a collection of metadata bits.
- The PTEs are stored in a (nested) array-like structure, such that a VPNs can be used as indices into the structure, pointing at the PTE that contains their corresponding PFN.
- The metadata bits:
  1. Valid bit:indicates whether PTE is valid.
     - When a program starts running, all the (virtual) space between the heap and stack is unused and hence unmapped. Hence entries at pagetable [VA] for those addresses are invalid.
     - If a program tries to access invalid memory, a trap is generated and the program is likely terminated.
     - The valid bit is crucial for supporting **sparse address spaces**.
     - Invalid PTEs don't have any physical memory mapped to them, this lets us use only the physical memory we need.
  2. Protection bits: indicate whether the page can be read from, written to or executed from. Invalid accesses generate a trap to the OS.
  3. Present bit: indicates whether page is present in memory (or has been swapped out to disk).
  4. Dirty bit: indicates whether the page has been modified since being loaded into memory.
  5. Reference (a.k.a. accessed) bit: used to track if the page has been accessed since the bit was last reset. This information helps in page replacement, to keep popular pages around.
  6. User/Supervisor bit: indicates if processes can access the page in user mode.
  7. Hardware caching bits: help determine how hardware caching works for the page.

How It Works

When the OS reads a command to access/write to a memory location at `VA0`,

1. Extract `VPN` and `offset` from `VA0`.
2. Access **PTE** stored at `PTBR + VPN`. **Memory Access 1 (PTE read)**
3. Extract the `PA` and **permission bits** from **PTE**.
4. Verify **permission bits**.
   - If page is invalid, raise `Segmentation Fault` exception.
   - If permissions are insufficient, raise `Protection Fault` exception.
5. Read/write to data stored at `PA + offset`. **Memory Access 2 (Data access)**

Making Paging Faster

The procedure given above is without the use of TLBs, which are always used in the real world to speed up paging.

Reducing The Page Table's Size

1. Increase Page Size: the factor by which the page size is increased is the factor by which the page table's size is reduced.
    - This approach is commonly used by DBMSs and other high-end commercial applications. However, their goal is to reduce pressure on the TLB by reducing the number of translations, instead of minimizing the page table's size.
    - The major problem with this is internal fragmentation. Hence, it's only suitable if the programs are guaranteed to use the large pages allocated to them almost thoroughly.
2. Hybrid approach: Paging + Segments
    - We first note that in each table, most of the PTEs are invalid, and only a select few are mapped to physical page frames.
    - This approach involves maintaining a separate page table for the code, stack and heap segments of the virtual address space.
    - We have base and bound registers for each segment as before, except that the **base** points to the physical address of the page table of that segment.
    - Note that all six registers must be updated on a context switch.
    - The translation involves:

```
V2P(VA v){
    segment = v[0:2];//or bit operations.
    VPN = v[2:k];//or bit operations.
    offset = v[k:];//or bit operations.
    addressOfPTE = Base[segment] + (VPN*sizeof(PTE));
    PTE = read(addressOfPTE);
    PFN = PTE[:c]
    PA =  PFN + offset;
    return PA;
}
```

    - However, like segmentation, (i) this method can't help with the wastage of memory due sparse usage within segments (ex. a sparsely used heap), and (ii) leads to external fragmentation, as the variable sized page tables need to be allocated contiguous chunks of memory.
3. Multi-level page tables:

Advantages of paging

- Flexibility is an important advantage of paging: it works regardless of how the process uses the address space, how the heap and stack grow or how they are used.
- Flexibility also ensures it can handle sparse address spaces and minimize internal fragmentation.
- Simplicity: when new pages are requested, the OS simply traverses a free list and returns the first pages that it finds.
- It avoid external fragmentation altogether by having fixed-size units.

4.2.4 TLB

- It's a part of the MMU and is a hardware cache of popular V2P translations.
- The process of memory access with the TLB is:
  1. Extract `VPN` and `offset` from `VA0`.
  2. If TLB has an entry for `VA0` (TLB hit)
     - If permissions are insufficient, raise `Protection Fault` exception.
     - Extract `PA` for `VA` from TLB.
     - Read/write to data at `PA + offset`. **Memory Access 1 (Data access)**
  3. Otherwise, carry out steps 2, 3 and 4 of the usual access routine. **Memory Access 1 (PTE read)**
  4. Instead of step 5, insert the translation `(VPN,PA,Protection Bits)` into the TLB.
  5. Retry the memory access instruction (start at 1 of this sequence).
- Like most caches, TLB is built on the premise that most translations are found in the cache (hits). The little overhead costs of using a TLB (little because it's near the processing core and designed to be quite fast), are made up for by the hit cases.
- The speed of a cache comes partly from its minimal size; any large cache is by definition slow.

TLB Misses

- We say a program exceeds the TLB coverage if it generates a large number of TLB misses, by accessing (in a short period of time) more pages than the number of translations the TLB can fit.
- One solution to this is to allow larger pages for storing key data structures that such programs access repeatedly, and thus allowing a single translation in the TLB to serve many memory accesses.
- Support for large pages is often exploited by programs such as DBMSs, which have large and randomly accessed data structures.
- In CISC computers, the hardware would handle the TLB miss entirely (low trust on the OS designers).
- With physically-indexed caches, some translation of the VA has to take place for the lookup itself, and thus they are very slow. Check cache notes if any.
  On a miss,
  - The hardware would walk the page table using `PTBR` to get the `PA` for the `VPN` that induced a miss.
  - It would update the TLB with the new translation.
  - Then hit retry on the instruction that resulted in a TLB miss.
- In RISC computers, the TLB is software-managed. On a TLB miss, the hardware just raises an exception (`TLB_MISS`).
- With software-managed TLBs, it is important that the instructions provided by the hardware to read/write to the TLB, are only allowed to be run in privileged modes.
  1. This pauses the current instruction stream, raises the privilege level to kernel mode and jumps to a **trap handler**.
  2. The targeted code looks up the translation in the page table, uses **privileged** instructions to update the TLB, and returns from the trap.

  Note that:
  - Note that on return the execution must pick up **at the causal instruction** that caused the trap, instead of the usual return-from-trap where execution picks up **after the causal instruction.**

  - While accessing the code for trap handler for `TLB_MISS`, the OS should not incur a TLB miss, which would imply an infinite chaing of TLB misses. To avoid this, we can:
    1. Keep TLB miss handlers in **unmapped** physical memory, so that their execution does not involve address translation (and hence TLBs).
    2. Reserve entries in TLB for permanently-valid (**wired**) translations and use some of these slots for the trap-handler code's memory address' translation.

- The advantage of a software-managed TLB include:
  - Flexibility: the OS can store the PTEs in any data structure it wants, as it's allowed to update the page table walk function, since it's not burned into a chip.
  - Simplicity: the hardware just has to raise an exception and relies on the OS to resolve the TLB miss.

TLB Contents

- The TLB has 32/64/128 entries and is **fully associative** ($\equiv$ any translation can be anywhere in the TLB, and the hardware searches the entire TLB in parallel to find the entry for `VPN`).
- The TLB entry has: VPN | PFN | other bits.
- The other bits are:
  - Valid bit: denoting if **the TLB entry is valid, NOT the page.**
  - Protection bits.
  - Global bit, to indicate globally shared translations. If this is set, ASID bit is ignored.
  - Address-space identifier (ASID) (to identify if the entry is for the given process or in a shared region).
  - Dirty bit, to indicate if the page has been modified since it was brought into memory.
  - Coherence bits: they determine how a page is cached by the hardware.
  - Page mask field: to support multiple page sizes.
- On a context switch, the OS must ensure one processes translations in the TLB are not used by another process (unless it's a shared memory region). This can be done via:
  1. Simply flushing the TLB on a context switch (marking each entry as invalid). This can be coded up in the OS or the hardware can be set to flush the TLB whenever the PTBR is updated.
  2. Maintain ASID for each entry and a privileged register to hold the ASID of the current process. Note that ASIDs require much fewer (8ish bits) as compared to PIDs (32ish bits).
     Additionally, ASIDs (along with the permission bits) enable sharing of code pages across processes.

Replacement Policy

We want to maximize the hit-rate in the TLB based on our choice of a replacement policy. Some common choices are:

1. Evict Least Recently Used entry: relies on temporal locality of address usage. The least recently used entry in the cache will probably not be used any time soon, as future access will be closer to the more recent accesses in the cache.
2. Random: Avoids making corner-case worst-possible decisions all the time. (For example, a for loop which requires TLB eviction, with the LRU policy in place.)

4.3 Free Space Management

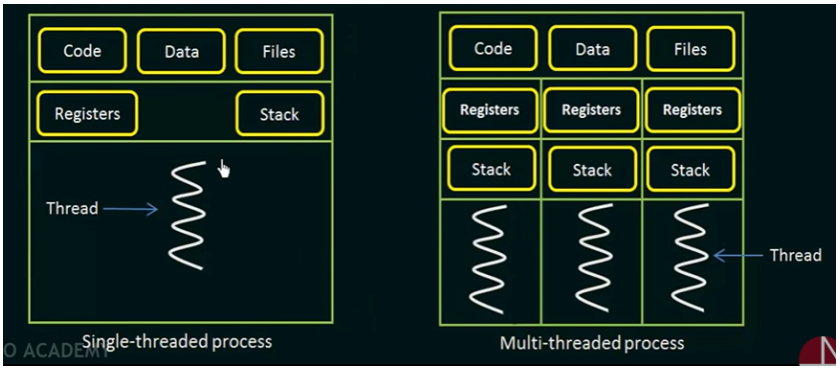
Chapter 5

Concurrency

From Neso Academy's OS lectures.

5.1 Basics of Threads

- Each program may have multiple processes associated with it. Each process may have multiple threads; threads are a basic unit of execution or CPU utilization.
- A thread comprises of:
  1. A thread ID
  2. A program counter
  3. A register set
  4. A stack.
- Threads of the same process share:
  1. Code section
  2. Data section
  3. Other OS resources, such as files and signals.
- A **traditional/heavyweight** has a single thread of control. Having multiple threads of control allows a process to perform multiple tasks at a time.



- The benefits of multithreaded programming are:
  1. **Responsiveness**: interactive applications allow a program to continue running even one of its tasks is blocked in its thread in a lengthy operation, thereby allowing responsiveness.

2. **Resource Sharing**: Code, data and files are shared between threads, allowing for better utilization of system resources.
3. **Economy**: Saves the costs of allocating a separate process for each task.
4. **Utilization of Multiprocessor Arch.**: Multithreading on a multi-CPU machine increases concurrency by using parallelism.

5.2 Multithreading Models and Hyperthreading

- There are two types of threads:
  1. User threads: Supported above the kernel and are managed without kernel support.
  2. Kernel threads: Supported and managed directly by the OS.
- Multithreading models describe the type of relations between user and kernel threads. There are three common types:
  1. Many-to-One
  2. One-to-One
  3. One-to-Many

5.2.1 Many-to-One

- Many user-level threads are mapped to a single kernel thread.
- Thread management is done by the thread library in user space; thus, it is efficient.
- The limitations are:
  1. The entire process will block if a thread makes a system call.
  2. Since only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiple processors.

5.2.2 One-to-One

- A bijection between user-level and kernel-level threads.
- Provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- Also allows multiple threads to run in parallel on multiple processors.
- The limitations are:
  1. Creating a user thread requires creating a corresponding kernel thread, which is costly.
  2. The overhead of creating kernel threads can burden the performance of an application; most implementations of this model restrict the number of threads supported by the system.

5.2.3 Many-to-Many

- Each user thread is mapped to many shared kernel threads. # kernel threads $\leq$ # user threads.
- The number of kernel threads may be specific to a particular application or machine.
- Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- Blocking system calls result in the kernel scheduling another thread for execution.
- This model makes the best of the previous two models and is what is implemented in most systems.
- However, with its difficulty of implementation and increasing number of cores, one-to-one is increasingly preferred.

5.3 Hyperthreading a.k.a. Simultaneous Multithreading (SMT)

- Multiple multithreaded programs running at the same time.
- Hyperthreaded systems allow their processor cores' resources to become multiple logical processors for performance.
- In hyperthreading, physcial cores are logically divided into multiple virtual cores, to support multiple threads running concurrently.
- It enables the processor to execute two (or more) threads, sets of instructions, at the same time.
- Hyperthreading allows two streams to be executed concurrently, it s like having two separate processors working together.

5.4 Process Synchronization

- A cooperating process is one that can affect or be affected by other processes executing in the system (as opposed to the usual processes that have certain guarrantees of isolation).
- Cooperating processes may
  - directly share a logical address space (code and data)
  - or be allowed to share data only through files or messages.
- Concurrent access to shared data may result in data inconsistency.
- Up ahead we discuss mechanisms to ensure the orderly execution of cooperating processes that share a logical address space ensuring data consistency.
- For sharing of data through buffers, the buffers may be:
  1. Bounded: The consumer must wait if it's empty, the producer must wait if it's full.
  2. Unbounded: The consumer must wait if it's empty, the producer never has to wait.
- Buffers are suplemented with a `counter` variable to track the number of items in the buffer.
- The possibly concurrent updates to the `counter` variable is a race condition.
- Race condition: A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the accesses take place.
- One of the goals of process synchronization (primitives) is to avoid race conditions.

5.4.1 Critical Sections

- If a system has n process, $P_1, P_2, ...P_n$
- Each process has a segment of code, called its **critical section**, where the process may be changing common (shared) variables, updating a table, writing to a file or accessing some other shared resource.
- No two processes should be allowed to execute their critical sections at the same time.
- The **critical-section problem** is to design a protocol that the processes can use to cooperate.
- Rules about critical sections:
  - Each process must request permission to enter its critical section.
  - The section of code implementing this request is the **entry section**.
  - The critical section must be followed by an **exit section**, where the process announces its exit from its critical section.
  - The remaining code is the **remainder section**.
- A solution to the critical section problem must satisfy:
  1. Mutual exclusion: No two processes can execute their critical sections simultaneously.
  2. Progress (liveness): If no process is executing its critical section, and multiple processes request to enter their critical sections, the decision of whom to permit must be taken by processes that are not executing in their remainder section. Additionally, the decision cannot be postponed indefinitely.
  3. Bounded waiting: There exists a limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before its request has been granted.

5.4.2 Peterson's Solution to the CSP

- Classic, software-based solution to the CSP.
- May not work correctly on modern computer architectures.
- Provides a good algorithmic description of solving the CSP, and illustrates relevant complexities of software that addresses the requirements of CSP.
- Solution outline:
  - The solution is restricted to two processes P_1 and P_2 that alternate execution between their critical and remainder sections.
  - Two variables need to be shared between P_1 and P_2:
    1. `int turn`: indicates whose turn it is to enter their critical section. It can be modified by both the processes.
    2. `bool flag[2]`: Used to indicate if a process is ready to enter its critical section.
  - Key Idea: P_i is ready to enter its critical section $\implies$ set `flag[i]` to true and set `turn` = j, $j \neq i$.
  - It's a humble algorithm.
- The algorithm:

```
int turn;
bool flag[2];
void process(int i){//i = index of process
    do{
        flag[i] = true;
        turn = j;//~ turn = otherProcessIndex();
        //j != i ~ assume other process is using its CS
```

```
        while(flag[j] && turn==j);
        //PC here => flag[j]=false || turn!=j
        //=>j is not at CS or it's not j's turn => my turn.
        doCriticalTasks();
        flag[i] = false;//Not in CS anymore.
        doRemainderTasks();
    }
    while(true);
}
```

5.4.3 Test and Set Lock

- Hardware-based solution to the CSP.
- There is a shared lock variable, which can take two values, 0 (unlocked) or 1 (locked).
- Before enterring the critical section, the process inquires about the lock.
- If it's already locked (some other process is in its CS), the process waits until it becomes free.
- If it's not locked, the process locks it and executes its CS.
- A great analogy for the lock is that processes can execute their CSs in a single, fixed room, whose door they lock on enterring and unlock on exiting (having finished their CS).
- **Atomic operation:** can not be interrupted, executes as a single operation; all or nothing.
- Note: testAndSet is an **atomic operation** equivalent to the following:

```
bool testAndSet(bool *target){
    bool rv = *target;
    *target = true;
    return rv;
}
```

- Processes use the function as follows:

```
bool lock;
bool testAndSet(bool* lock);
void process(int i){
    do{
        while(testAndSet(&lock));
        //acquired lock
        doCriticalTasks();
        lock = false;
        doRemainderTasks();
    }
    while(true);
}
```

- This solution satisfies mutual exclusion and progress requirements.
- It is **susceptible to starvation** with more than 2 processes; it doesn't satisfy the bounded waiting condition.

5.4.4 Semaphores

- Software-based solution to the CSP.
- They implement a management of concurrent processes usinga single integer value, called a semaphore.
- A semaphore is a non-negative variable which is shared between threads, to achieve process synchronization.
- A semaphore `S` only allows for:

1. Initialization to an integer (non-negative).
2. `wait()` aka `P()`: atomic operation. (**probe**ren, to test.)
3. `signal()` aka `V()`: atomic operation. (verhogen, to increment.)

- Wait() (atomic) operation:

```
P(Semaphore S){
    while(S<=0);
    S--;//no problem of inconsistency, since atomic
}
```

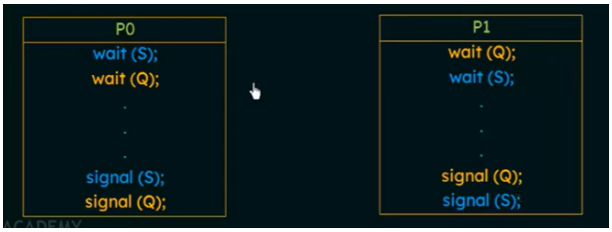
- Wait (P) is called before enterring a critical section.
- Singal() (atomic) operation:

```
V(Semaphore S){
    S++;//no problem of inconsistency, since atomic
}
```

- Singal (V) is called on exiting a critical section.
- To reiterate, both P and V are atomic operations; no two processes can modify `S` simultaneously.
- Types of semaphores:
  1. Binary Semaphores: S is either 0 or 1. These are aka mutex locks, as they are locks that provide mutual exclusion. It is initialized to 1.
  2. Counting Semaphore: The value ranges over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Disadvantages of Semaphores

- Main disadvantage: busy waiting inside the P call.
  - Busy wait: While a process is in its CS, any other process that tries to enter its CS must loop continuously in its entry section. This consumes (wastes) CPU cycles that other processes may have used productively.
  - These are spinlock semaphores, as the process "spins" while waiting for the lock.
- To overcome the busy waiting disadvantage, we can make slight modifications to P and V.
- New P function:
  - Rather than engaging in busy waiting, the process cna block itself.
  - The block operation puts the process into a waiting queue associated with the semaphore, and the state of the process is switched to a waiting state.
  - The control is transferred to the CPU scheduler, which selects another process to execute.
- New V function:
  - Send a signal to the queue to pop a blocked process. If the queue is empty, increment S instead.
- Deadlock and Starvation:
  - The implementation with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the two processes. This requires at least two semaphores, used by both the processes.



5.4.5 Bounded Buffer Problem, aka Producer-Consumer Problem

- There is a buffer of n slots, where each slot can store one unit of data.
- There are two processes accessing the buffer:
  1. Producer: inserts data into (empty slots in) the buffer.
  2. Consumer: removes data from (non-empty slots in) the buffer.
- Requirements for sound functioning:
  1. The producer must not insert data when the buffer is full.
  2. The consumer must not remove data when the buffer is empty.
  3. The removal and insertion of data shouldn't happen simultaneously.
- The solution involves three semaphores:
  1. m (mutex): binary semaphore used to acquire and release a common lock on the buffer.
  2. empty: a counting semaphore, initialized to the buffer size. It represents the number of empty slots in the buffer.
  3. full:a counting semaphore, initialized to 0; it represents the number of full semaphores in the buffer.
- The producer code is:

```
void producer(){
do{
    wait(empty);//ensure empty slots exist, decrement by 1
    wait(mutex);//acquire lock.
    addDataToBuffer();
    signal(mutex);//release lock.
    signal(full);//increment full.
} while(true);
}

void consumer(){
do{
    wait(full);//ensure data exists, decrement by 1.
    wait(mutex);//acquire lock
    removeDataFromBuffer();
    signal(mutex);//release lock
    signal(empty);//increment empty
} while(true);
}
```

5.4.6 Readers-Writers Problem

- A database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database (readers), whereas other may want to read and write to it (writers).

- Inconsistencies arise when a writer and any other thread (writer or reader) access the db simultaneously.
- To avoid these, writers must have exclusive access to the database.
- We use two semaphores and an integer value:
  - int readCount; tracks number of readers in the critical section.
  - Mutex: initialized to 1, used to ensure mutual exclusion between accesses to readCount.
  - writer: initialized to 1, used by writer and reader processes.
- Code:

```
Semaphore writer(1), mutex(1);
int readCount = 0;
void writer(){
do{
    wait(writer);

    doWriteTasks();

    signal(writer);
}while(true);
}
void reader(){
do{
    wait(mutex);
    readCount+=1;
    if(readCount==1){
        wait(writer);
    }
    signal(mutex);

    doReadTasks();

    wait(mutex);
    readCount-=1;
    if(readCount==0){
        signal(writer);
    }
    signal(mutex);

}while(true);
}
```

5.4.7 The Dining Philosophers Problem

- N (say 5) philophers at a circular table, with N forks placed between each pair of neighbours.
- Each philosopher needs two forks to eat his meal, whenever he wants to eat.
- Once done, he places the forks back down, now available to his neighbours.
- Naive code:

```
Semaphore forks[N](1);//initalized to 1.
void tryEating(int i){//i = philopher's index.
do{
    wait(forks[i]);
    wait(forks[i+1 % N]);
    eatFood();
    signal(forks[i]);
    signal(forks[i+1 % N]);
    think();
}
while(true);
}
```

- This is susceptible to a deadlock when all philosophers get hungry at the same time and grab fork[i], stuck on waiting for fork[i+1].
- Possible solutions:
  - Allow an extra fork, or reduce the number of philosophers by 1.
  - Allow a philosopher to pick up both chopsticks, or neither. $\implies$ The pick-up becomes a critical section.
  - Asymmetric solution: pick i or i+1%N first, depending of the value of i%2.

5.4.8 Monitors

- A high level abstraction that provides a convenient and effective mechanism for process synchronization.
- A monitor type presents a set of programmer-defined operations that provide mutual exclusion within the monitor.
- The monitor type also contains **shared** variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables, with care for synchronization.
- The monitor enforces that its local variables can be accessed only by its public procedures, not directly; they are private.
- The monitor construct ensures that only one process at a time can execute its procedures.
- Monitors require a new construct, called a Condition variable. Condition variables (say, `cond1`) have methods wait() and signal().
  - `cond1.wait()` means the caller process is suspended until another process invokes `cond1.signal()`
  - `cond1.signal()` resumes **exactly one** suspended process.
- The monitor also has an entry queue, where processes waiting to use its procedures are stored.
- Additionally, each condition variable in the monitor has its own separate queue of suspended processes.

Dining Philosophers Problem using Monitors

- We impose the restriction that a philosopher may pick up his forks only if both of them are available.

```
enum {thinking,hungry,eating} state[5];
/*
state[i] can be set to eating, only if both state[(i+4)%5] and state[(i+1)%5] are not eating.
*/
condition self[5];//vars to hold hungry state, signal fork down.
//using i-1, i+1 for brevity.
void test(int i){
    if((state[i-1]!=eating && state[i+1]!=eating) && state[i]==hungry){
        state[i] = eating;
        self[i].signal();
    }
}
void pickup(int i){
    state[i] = hungry;
    test(i);
    if(state[i]!=eating){
        self[i].wait();
    }
}
void putdown(int i){
    state[i] = thinking;
    test(i+1);
    test(i-1);
}
```

5.5 Notes from Interview Questions

Chapter 6

Misc.

Chapter 7

References

1. OSTEP