OPERATING SYSTEMS

Paging

The major role of the page table is to store address translations for each of the virtual pages of the address space, thus letting us know where in physical memory each page resides. Let's imagine the process with that tiny address space (64 bytes) is performing memory access:

To translate this virtual address that the process generated, we have to first split it into two components: the virtual page number (VPN), and the offset within the page. 32-bit address space, with 4KB pages i.e. 12 bits for offset. A 20-bit VPN

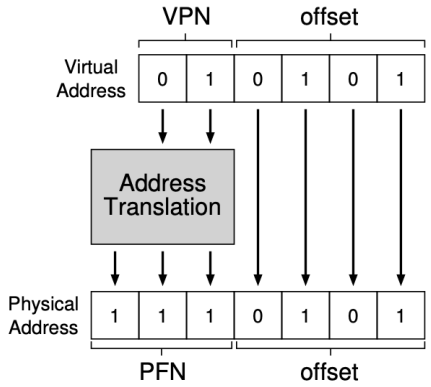


Figure 18.3: **The Address Translation Process**

implies that there are 2^{20} translations that the OS would have to manage for each process

If PAS is greater than the VAS, that means, if your main memory can store more processes than Hard disk, then use PAS for Page table Size mapping for processes.

4 bytes per page table entry (PTE) to hold the physical translation plus any other useful stuff, we get an immense 4MB of memory needed for each page table.

A page table is just a data structure that is used to map virtual addresses (or really, virtual page numbers) to physical addresses (physical frame numbers).

A valid bit is crucial for supporting a sparse address space; by simply marking all the unused pages in the address space invalid, we remove the need to allocate physical frames for those pages and thus save a great deal of memory.

A dirty bit is also common, indicating whether the page has been modified since it was brought into memory.

Once this physical address is known, the hardware can fetch the PTE from memory, extract the PFN, and concatenate it with the offset from the virtual address to form the desired physical address

A TLB is part of the chip's memory-management unit (MMU), and is simply a hardware cache of popular virtual-to-physical address translations.

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                   // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
    else
        TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
        RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

First, extract the virtual page number (VPN) from the virtual address. and check if the TLB holds the translation for this VPN If it does, we have a TLB hit, which means the TLB holds the translation. Success! We can now extract the page frame number (PFN) from the relevant TLB entry, concatenate that onto the offset from the original virtual address, and form the desired physical address (PA), and access memory (Lines 5–7), assuming protection checks do not fail (Line 4).

VPN | PFN | other bits

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

TLB contents

During Context Switch, TLB can either be flushed, or address space identifier is maintained. By flushing the TLB on each context switch, we now have a working

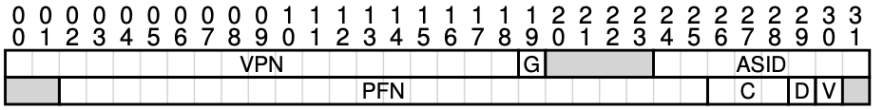
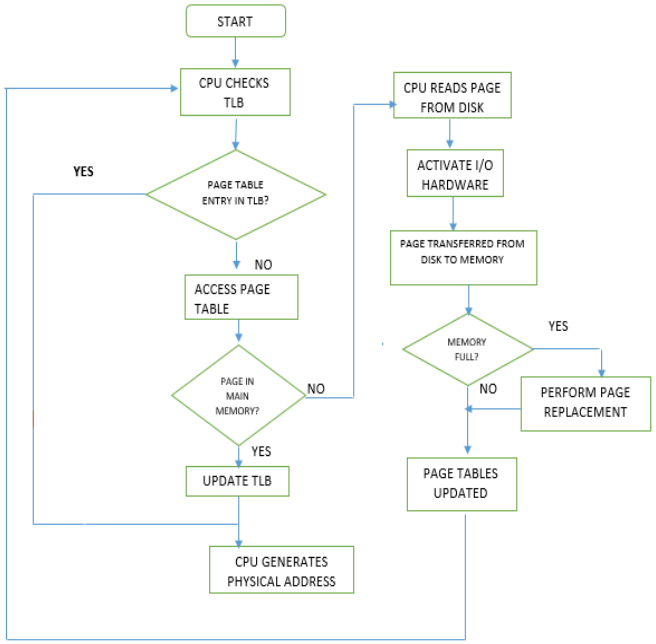


Figure 19.4: **A MIPS TLB Entry**

solution. However, there is a cost: each time a process runs, it must incur TLB misses as it touches its data and code pages. If the OS switches between



processes frequently, this cost may be high.

Page Size increases by a factor of X, then Page table size decreases by a factor of x
Q. Then why do we need multi-level paging? If we can always increase page sizes?
A. big pages lead to waste within each page, a problem known as internal fragmentation (as the waste is internal to the unit of allocation).
how to get rid of all those invalid regions in the page table instead of keeping them all in memory: Segmented Paging fails here. We will use a multi-level paging.

Multilevel Paging
How does it work?
If our page table size is greater than the Page frame number bits, we need to further divide the page table into number of pages, and accordingly do multilevel paging.
Suppose we have 32 bits as VAS and 36 bits as PAS and PS is 12 bits and PTE is 4B
Page frame number bits = 36-12 = 24
Page Number bits = 32-12 = 20
If your Virtual Memory is organized as 2,9,9,12
This means that the Outer level page Table has 4 entries. PTS for outer level Page is 4*4B = 16B
PTS for second level : 512*4B = 2048B = 11bits
PTS for third level : 512*4B = 2048B = 11 bits
Now, the third-level page table has how many PAGE table entry bits? 36-11 = 25 , likewise for second-level page table i.e. 25
And the first level page table will always require 24 bits as that is the Page frame no bit size.

Virtual Memory
If the VPN is not found in the TLB (i.e., a TLB miss), the hardware locates the page table in memory (using the page table base register) and looks up the page table entry (PTE) for this page using the VPN as an index. If the page is valid and present in physical memory, the hardware extracts the PFN from the PTE, installs it in the TLB, and retries the instruction, this time generating a TLB hit; so far, so good.
If we wish to allow pages to be swapped to disk, however, we must add even more machinery. Specifically, when the hardware looks in the PTE, it may find that the page is not present in physical memory. The way the hardware (or the OS, in a software-managed TLB approach) determines this is through a new piece of information in each page-table entry, known as the present bit. If the present bit is set to one, it means the page is present in physical memory and everything proceeds as above; if it is set to zero, the page is not in memory but rather on disk somewhere. The act of accessing a page that is not in physical memory is commonly referred to as a page fault.

Synchronization
Making two things happen at the same time is known as synchronization
In computer science, it refers to relationships among events. (before, during, after)

Synchronisation Constraints:
Serialization- Event A must happen before Event B.
Mutual Exclusion- Event A and B cannot happen at the same time

How a computer Program runs: The computer executes one instruction at a time in a sequence.

Concurrency: An Introduction

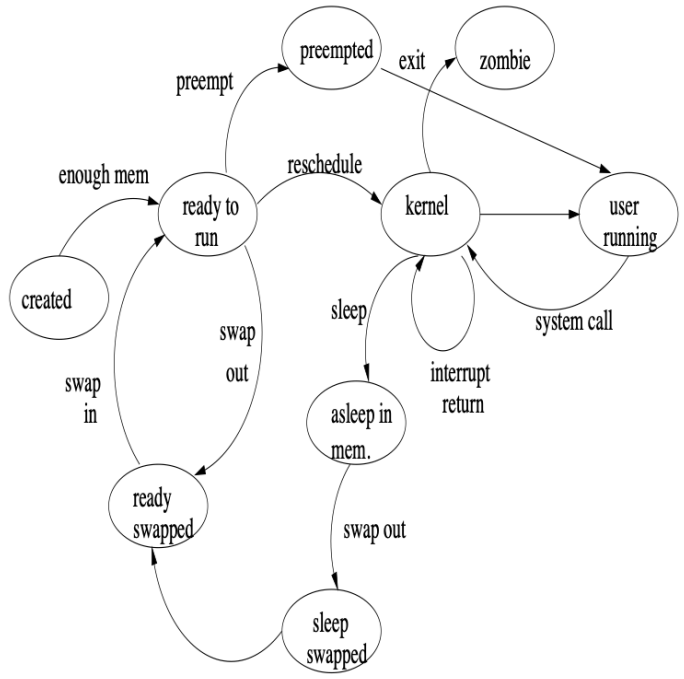
A thread is just like a process as each process has a Program Counter associated with it, but in the case of a thread, they share the same address space and thus can access the same data. We need one Thread Control Block for processes that share the same address space.
During context switch in a thread, as the address space remains the same, so there is no need to replace the page table.
As each thread runs independently, there may be recursion involved, so each thread has its own stack space, which it does not share with the other threads which are sharing the same data space.

Why do we use threads?
To achieve parallelism
To avoid blocking program progress due to slow I/O

A critical section is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.
What we really want for this code is what we call mutual exclusion. This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.
A race condition arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.

- The fork system call returns 0 to child and any integer > 0, to the parent. On fork the entire code is cloned in the child. **If-part** is executed by the parent after fork, and **else-part** is executed by the child. The parent makes a=60 and closed file descriptor. The child however has a=50, which is printed after execution. As the file descriptor (fd) is also copied during the fork, the child can use it even though the parent has closed it.
- CPU and I/O burst alternate.
- FCFS + pre emption = Round Robin
- **Multilevel queue scheduling with feedback is used in UNIX.**

UNIX process state diagram



- Unix Kernel has three basic jobs:
  - Create User Processes and Schedule their execution
  - Provide with sysem services
  - Service hardware interrupts and exceptions
-
- A system call definitely will context switch
- Semaphores can be used to implement scheduling constraints.
- Mutex performs P(x) and V(x) on binary semaphores.
- Enable and disable interrupts work on a single CPU and disables Context switching within the critical section.
- When a set of concurrent processes are in a deadlock situation, the degree of multiprogramming in the system decreases.
- The page table must be updated as soon as the frame address is changed.
- The second chance algorithm degenerated to the FIFO when all the page frames are in use.
- The degree of multiprogramming is limited by the size of the RAM
- Progress is not achieved when one process does not want to enter the critical section problem.
- The process of assigning load addresses to the various parts of the program and adjusting the code and data in the program to reflect the assigned addresses is called Relocation

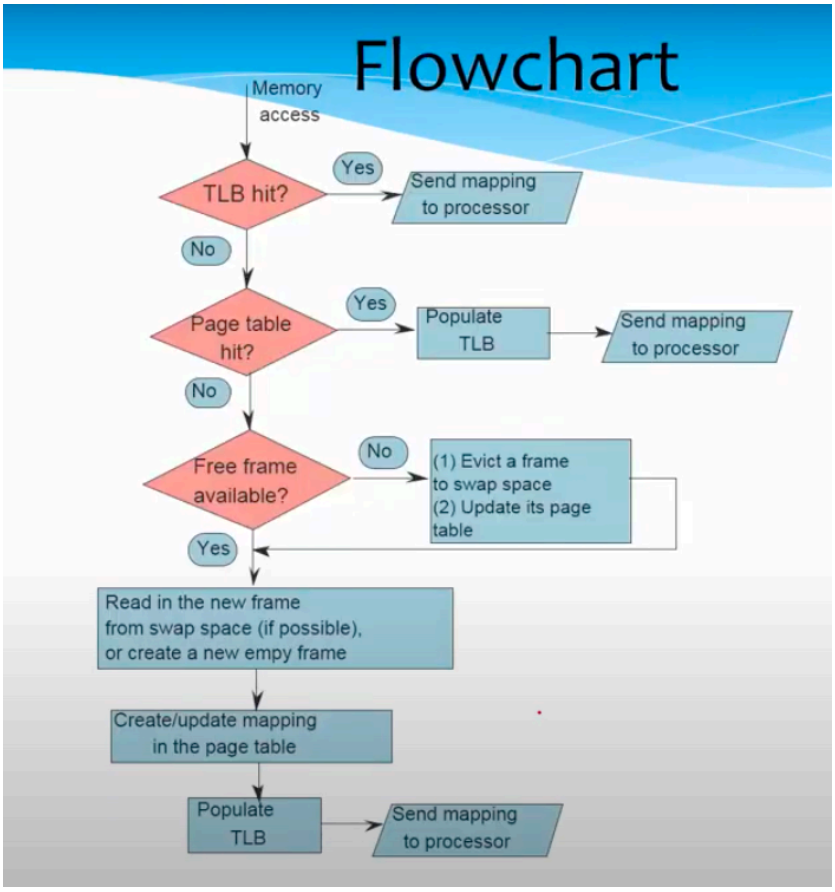
A hardware interrupt is not really part of CPU multitasking, but may drive it.

1. Hardware interrupts are issued by hardware devices like disk, network cards, keyboards, clocks, etc. Each device or set of devices will have its own IRQ (Interrupt ReQuest) line. Based on the IRQ the CPU will dispatch the request to the appropriate hardware driver. (Hardware drivers are usually subroutines within the kernel rather than a separate process.)
2. The driver which handles the interrupt is run on the CPU. The CPU is interrupted from what it was doing to handle the interrupt, so nothing additional is required to get the CPU's attention. In multiprocessor systems, an interrupt will usually only interrupt one of the CPUs. (As a special cases mainframes have hardware channels which can deal with multiple interrupts without support from the main CPU.)
3. The hardware interrupt interrupts the CPU directly. This will cause the relevant code in the kernel process to be triggered. For processes that take some time to process, the interrupt code may allow itself to be interrupted by other hardware interrupts.

In the case of timer interrupt, the kernel scheduler code may suspend the process that was running and allow another process to run. It is the presence of the scheduler code which enables multitasking.

Software interrupts are processed much like hardware interrupts. However, they can only be generated by processes which are currently running.

1. Typically software interrupts are requests for I/O (Input or Output). These will call kernel routines which will schedule the I/O to occur. For some devices the I/O will be done immediately, but disk I/O is usually queued and done at a later time. Depending on the I/O being done, the process may be suspended until the I/O completes, causing the kernel scheduler to select another process to run. I/O may occur between processes and the processing is usually scheduled in the same manner as disk I/O.
2. The software interrupt only talks to the kernel. It is the responsibility of the kernel to schedule any other processes which need to run. This could be another process at the end of a pipe. Some kernels permit some parts of a device driver to exist in user space, and the kernel will schedule this process to run when needed.
   It is correct that a software interrupt doesn't directly interrupt the CPU. Only code that is currently running code can generate a software interrupt. The interrupt is a request for the kernel to do something (usually I/O) for the running process. A special software interrupt is a Yield call, which requests the kernel scheduler to check to see if some other process can run.



Virtually Indexed Physically Tagged Cache and page color concept

If your Set Index + Offset is greater than the Block offset of the RAM, then you need page colors.

Generally you cannot use VIPT cache if the Set index and block index of Cache exceeds the Block offset as we cannot extract the word from cache and get our tag ready for comparison before the virtual to physical address translation by the TLB.

Hence we need this page color. These are the bits which are left after we subtract Set index and block offset from the TAG of the Main memory.(Physically tagged)

Round robin scheduling behaves identically to FIFO if the job lengths are no longer than the length of the time slice. Round robin scheduling performs poorly compared to FIFO if the job lengths are all the same and much greater than the time slice length. S_2 is false. The dispatcher is only responsible for switching threads; the scheduler determines the thread priorities, based on the system's scheduling algorithms.

Bounded waiting can refer to "finite waiting time" as per many standard resources. And when no process executes for infinite time in CS, bounded waiting does not imply no starvation.

Shortest Job First (SJF) algorithm is optimal with respect to the average turn around time.
SJF algorithm is optimal with respect to the average waiting time.
SJF algorithm may lead to starvation.
SJF algorithm guarantees a better average turn around time than First Come First Served algorithm

If a system is in a safe state $\Longrightarrow$ no deadlocks
If a system is in an unsafe state $\Longrightarrow$ possibility of deadlock
Deadlock Avoidance $\Longrightarrow$ ensures that a system will never enter an unsafe state.

Deadlock prevention is more stricter than deadlock avoidance. In deadlock prevention, we need to ensure one of the four necessary conditions of deadlock does not occur. So, it may be the case that a resource request might be rejected even if the resulting state is safe. (One example, is when we impose a strict ordering for the processes to request resources).

Deadlock avoidance is less restrictive than deadlock prevention. Deadlock avoidance is like a police man and deadlock prevention is like a traffic light. The former is less restrictive and allows more concurrency. For the same reason, deadlock prevention ensures lesser resource utilization than deadlock avoidance. Option C is true and option D false.

If each resource type has exactly one instance, then a cycle in resource allocation graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle in the resource allocation graph is a necessary condition for the existence of deadlock but not sufficient. That is, even if there is a cycle, a deadlock may not be present.

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is know as a context switch.

Whenever an interrupt arrives, the CPU must do a state-save of the currently running process, then switch into kernel mode to handle the interrupt, and then do a state-restore of the interrupted process.

Processes may create other processes through appropriate system calls, such as fork or spawn. The process which does the creating is termed the parent of the other process, which is termed its child.

All processes are stored in the job queue. And processes in the Ready state are placed in the ready queue.

The space on the Stack is reserved for local variables when they are declared. These are static size (size of allocation is known even before the program run) allocations and are created and freed whenever these variables life starts and ends during a function call/return.

The Data section is made up of the global and static variables, allocated and initialized prior to executing the main. These are static size allocations and they are not freed until the program run finishes. Some of the Data section contents are R/W but some are Read Only (RO) like those used for string literals.

The Heap is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc. These are dynamically allocated and the size of allocation depends on the runtime values.

The Text section is made up of the compiled program code, read in from non-volatile storage when the program is launched. They are typically Read Only (RO).

(a, b)

I. Threads share:
- Address space
- Heap
- Static data
- Code segments
- File descriptors
- Global variables
- Child processes
- Pending alarms
- Signals and signal handlers
- Accounting information

II. Threads have their own:
- Program counter
- Registers
- Stack
- State

Option (d) is correct.

You should know about two things before solving this question

Indefinite Blocking (Starvation) vs Definite Blocking ..

Indefinite blocking means "we don't know how much time we should wait"

Definite blocking means "we know how much time we should wait"

So now think about the question

In all the cases we don't know how much time we should wait.

Point to recall :

FCFS Scheduling is starvation free because we know how much time we should wait(sum of burst times of all the process which came earlier)

Look at the Statement about FCFS :"Because there is no pre-emption, if a process executes for a long time, the processes in the back of the queue will have to wait for a long time before they get a chance to be executed."

This is not starvation, we are just waiting .....

Hope this will make sense....Thank you

A virtual memory system uses First In First Out (FIFO) page replacement policy and allocates a fixed number of frames to a process. Consider the following statements:
P : Increasing the number of page frames allocated to a process sometimes increases the page fault rate.
Q : Some programs do not exhibit locality of reference.

Both are true but one is not the cause of other