Syllabus:

Ch 1,2 mainly madam er slide e jaja topic ase ogulai
ch3 te process related ja ase except last er kisu example ase windows linux dia ogula baad

Ch4 e 4.3 porjonto shudhu

5 e multilevel feedback queue porjonto ar multicore system, pore example type ogula nai

6 e monitor ar 6.9 baade shob

7 e oi producer-consumer, reader-writer and dining philosopher

8 full

9 e 9.3 porjonto afaik

10 e khali page replacement algo gulai mone ase ar janina

Chapter 6

Race condition: A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().

A cooperating process is one that can affect or be affected by other processes executing in the system.

Critical Section:
The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.

A solution to the critical-section problem must satisfy the following three
Requirements:

1. Mutual exclusion. If process Pi is executing in its critical section, then no other processes can be executed in their critical sections.
2. Progress. If no process is executing in its critical section and some processes
wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. Bounded waiting. There exists a bound, or 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 that request is granted.

Peterson's Solution:

```
while (true) {
flag[i] = true;
turn = j;
while (flag[j] && turn == j)
;
/* critical section */
flag[i] = false;
/*remainder section */
}
```

computer architectures provide instructions that can force any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to
threads running on other processors. Such instructions are known as **memory barriers** or **memory fences**.

A mutex lock has a boolean variable available whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.

The definition of **wait()** is as follows:

```
 wait(S)
{ while (S <= 0) ;
// busy wait
S--; }
```

The definition of **signal()** is as follows:

```
signal(S)
{ S++; }
```

Liveness refers to a set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle. A process waiting indefinitely under the circumstances just described is an example of a “liveness failure.”

Chapter 8

Deadlock is a situation in which every process in a set of processes is waiting for an event that can be caused only by another process in the set.

livelock occurs when a thread continuously attempts an action
that fails.

A deadlock situation can arise if the following four conditions hold simultaneously
in a system:

1. **Mutual exclusion.** At least one resource must be held in a non sharable

mode; that is, only one thread at a time can use the resource. If another thread requests that resource, the requesting thread must be delayed until the resource has been released.

2. **Hold and wait.** A thread must hold at least one resource and waiting to acquire additional resources that are currently being held by other threads.

3. **No preemption.** Resources cannot be preempted; that is, a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.

4. **Circular wait.** A set {T0, T1, ..., Tn } of waiting threads must exist such that T0 is waiting for a resource held by T1, T1 is waiting for a resource held by T2, ..., Tn−1 is waiting for a resource held by Tn, and Tn is waiting for a resource held by T0.

Resource allocation Graph:

V(T, R)

E(Ti -> Rj, Rj ->Ti)

Ti -> Rj - Request Edge

Rj ->Ti - Assignment Edge

A claim edge Ti→Rj indicates that thread Ti may request resource Rj at some time in the future.

Pictorially, Ti - Circle

Rj - Rectangle

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no thread in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

Generally speaking, we can deal with the deadlock problem in one of three ways:

• We can ignore the problem altogether and pretend that deadlocks never occur in the system.
• We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
• We can allow the system to enter a deadlocked state, detect it, and recover.

Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold.

Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a thread will request and use during its lifetime

Safety Algorithm

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available and Finish[i] = false for i = 0, 1, ..., n − 1.
2. Find an index i such that both
a. Finish[i] == false
b. Needi ≤ Work
If no such i exists, go to step 4.
3. Work = Work + Allocation[i]
Finish[i] = true
Go to step 2.
4. If Finish[i] == true for all i, then the system is in a safe state.
This algorithm may require an order of m × n2 operations to determine whether a state is safe.

Resource-Request Algorithm

1. If Request[i] ≤ Need[i], go to step 2. Otherwise, raise an error condition, since

the thread has exceeded its maximum claim.
2. If Request[i] ≤ Available, go to step 3. Otherwise, Ti must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to thread Ti by modifying the state as follows:
Available = Available–Request[i]
Allocation[i] = Allocation[i] + Request[i]
Need[i] = Need[i] –Request[i]
If the resulting resource-allocation state is safe, the transaction is completed,
and thread Ti is allocated its resources. However, if the new state
is unsafe, then Ti must wait for Request[i], and the old resource-allocation state is restored.

Deadlock detection algo:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available. For i = 0, 1, ..., n–1, if Allocationi ≠ 0, then Finish[i] = false. Otherwise, Finish[i] = true.
2. Find an index i such that both
a. Finish[i] == false
b. Requesti ≤ Work
If no such i exists, go to step 4.
3. Work = Work + Allocation[i]
Finish[i] = true
Go to step 2.
4. If Finish[i] ==false for some i, 0 ≤ i < n, then the system is in a deadlocked
state. Moreover, if Finish[i] == false, then thread Ti is deadlocked.

There are two options for breaking a deadlock. One is simply to abort one or more threads to break the circular wait. The other is to preempt some resources from one or more of the deadlocked threads.

• **Abort all deadlocked processes.** This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.

• **Abort one process at a time until the deadlock cycle is eliminated**. This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Resource Preemption:

1. Selecting a victim
2. Rollback
3. Starvation

Chapter 9

Address Binding: Address Binding is the mapping of a physical address to a logical address known as a virtual address, it allocates a physical memory region to a logical pointer.

Compile Time Binding: Addresses in the source program are generally symbolic (such as the variable count). A compiler typically binds these symbolic addresses to relocatable addresses (such as “14 bytes from the beginning of this module”).

If you know at compile time where the process will reside in memory, then absolute code can be generated.

Load time Binding: If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time.
Execution time Binding: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

The execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address.**

Dynamically linked libraries (DLLs) are system libraries that are linked to user programs when the programs are run.

In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process.

Memory Protection:
When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

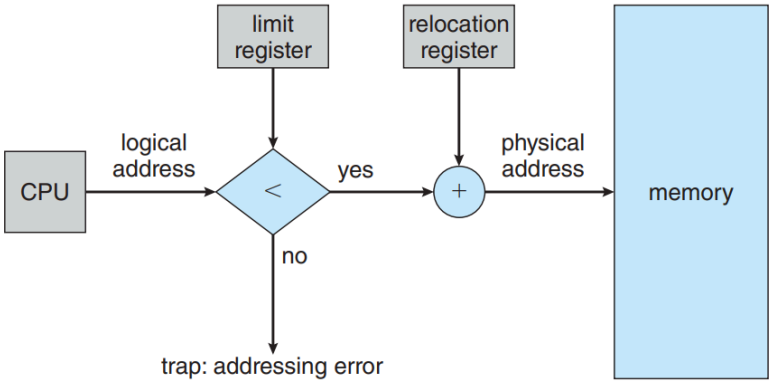

Figure 9.6 Hardware support for relocation and limit registers.

Variable Sized Partition:

- First fit- First Hole
- Best Fit- Smallest Hole
- Worst fit -> Largest Hole. (actually works best)

Variable Sized Partition -> No internal fragmentation, only external
Fixed Sized Partition -> Both internal and external fragmentation

Internal fragmentation is the unused memory that is internal to a partition. Occurs in Fixed Size partitioning when the allocated partition size is bigger than the process size.

External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes.
Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation.
Solution: Compaction, non-contiguous addressing (eg- paging)

Non Contiguous Memory Allocation -> Paging:

1. Breaking physical memory into fixed-sized blocks called frames.
2. Breaking logical memory into blocks of the same size called pages.

3. Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d)
4. The page number is used as an index into a per-process page table.
5. Steps taken by the MMU to translate a logical address generated by the CPU to a physical address:
    - Extract the page number p and use it as an index into the page table.
    - Extract the corresponding frame number f from the page table.
    - Replace the page number p in the logical address with the frame number f

Advantage: The possibility of sharing common code
Drawback: Internal Fragmentation
Translation look-aside buffer (TLB) is a special, small, fast-lookup hardware cache.
Working mechanism-

1. When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB.
2. Then either TLB hit or TLB miss can occur.

TLB Hit: If the page number is found, its frame number is immediately available and is used to access memory.

The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**.

TLB Miss: If the page number is not in the TLB a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory. In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.

- If the TLB is already full of entries, an existing entry must be selected for replacement.
- Some TLBs allow certain entries to be wired down, meaning that they cannot be removed from the TLB. Typically, TLB entries for key kernel code are wired down.

- Some TLBs store address-space identifiers (ASIDs) in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. ASID allows the TLB to contain entries for several different processes simultaneously. If the TLB does not support separate ASIDs, then every time a new page table is selected (for instance, with each context switch), the TLB must be flushed (or erased) to ensure that the next executing process does not use the wrong translation information.

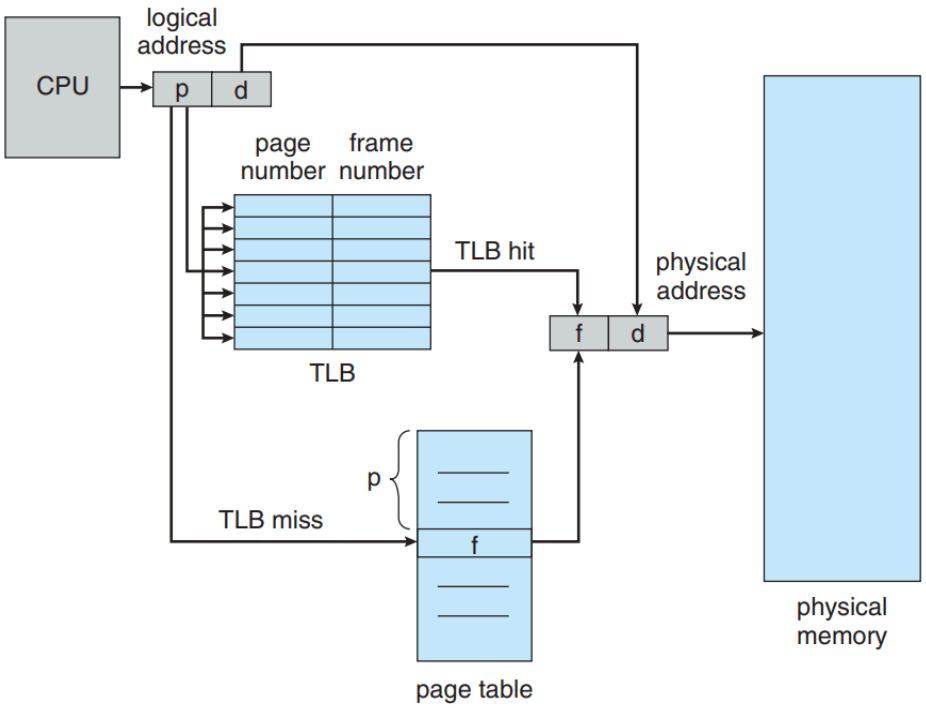

Figure 9.12 Paging hardware with TLB.

TLB Math: An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 10 nanoseconds to access memory, then a mapped-memory access takes 10 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (10 nanoseconds) and then

access the desired byte in memory (10 nanoseconds), for a total of 20 nanoseconds. (We are assuming that a page table lookup takes only one memory access, but it can take more, as we shall see.)

Solution: To find the effective memory-access time, we weight the case by its probability:

$$\text{effective access time} = 0.80 \times 10 + 0.20 \times 20$$
$$= 12 \text{ nanoseconds}$$

Chapter 10

Virtual memory is a technique that allows the execution of processes that are not completely in memory

The ability to execute a program that is only partially in memory would confer many benefits:
• A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large virtual address space, simplifying the programming task.
• Because each program could take less physical memory, more programs could be run at the same time,with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
• Less I/O would be needed to load or swap portions of programs into memory, so each program would run faster.

A **page fault** occurs when a program attempts to access a page that is not currently in physical memory, triggering the operating system to load the page from disk.

Effective access time without **page faults equals memory access time**.

With page faults, **effective access time = (1 - p) × ma + p × page fault time**, where p is the probability of a page fault.

Major Components of Page-Fault Service Time:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

Belady's anomaly: for some page-replacement
algorithms, the page-fault rate may increase as the number of allocated frames
increases.

Two implementations of **LRU** are feasible:

1. Counters
2. Stack