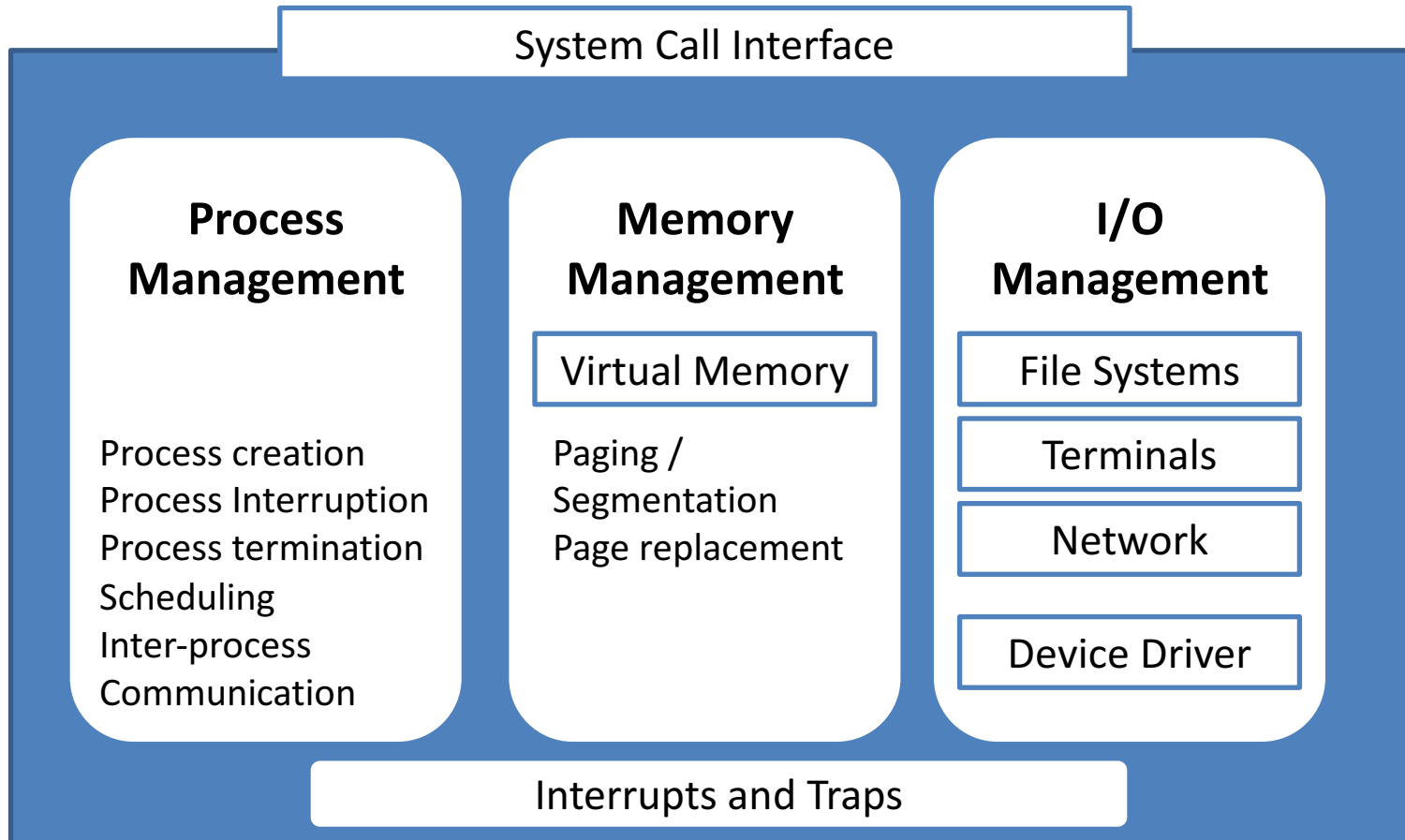


Revision

CS3026 Operating Systems

Lecture 23

Three Main Areas



Process Management

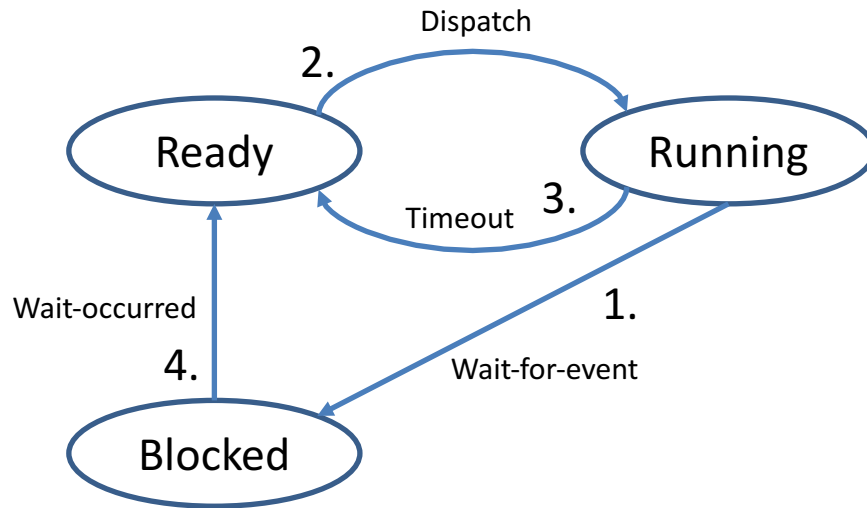
Process

- Definitions:
 - A process is a program in execution
 - The entity that can be assigned to and executed on a processor
- Processes are a fundamental concept of an operating system
 - As a unit of execution, it enables the concurrent execution of multiple programs on a system
 - CPU switches between processes
- Process is defined as the unit of resource ownership and protection

Process

- Process is defined as the unit of resource allocation and protection
 - Protected access to processors, other processes, files, I/O
- Processes embody two characteristic concepts
 - *Resource ownership*:
 - Processes own an address space (Virtual memory space to hold the process image)
 - Processes own a set of resources (I/O devices, I/O channels, files, main memory)
 - Operating system protects process to prevent unwanted interference between processes
 - *Scheduling / execution*
 - Process has execution state (running, ready etc.) and dispatching priority
 - Execution follows an execution path (trace) through one or more programs
 - This execution may be interleaved with that of other processes

Process Execution



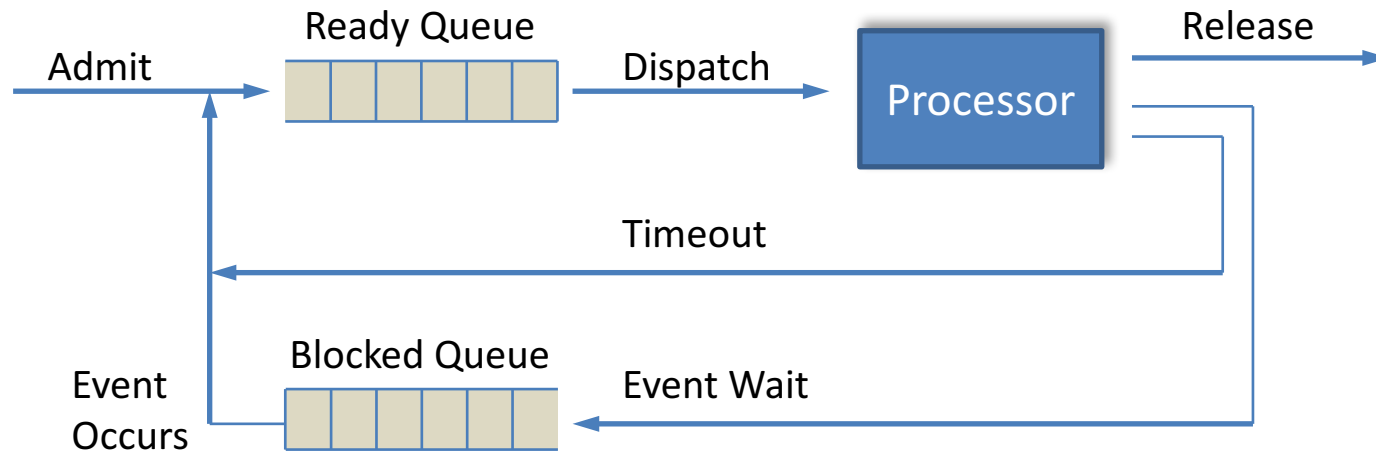
1. Process blocked for I/O
2. Dispatcher schedules another process
3. Dispatcher interrupts process because its time slice expired
4. Input becomes available, blocked process made ready

- We can distinguish three basic process states during execution
 - Running: actually using the CPU
 - Ready: being runnable, temporarily stopped (time-out) to let another process execute
 - Blocked/Waiting: unable to run until some external event happens, such as I/O completion

State Transitions

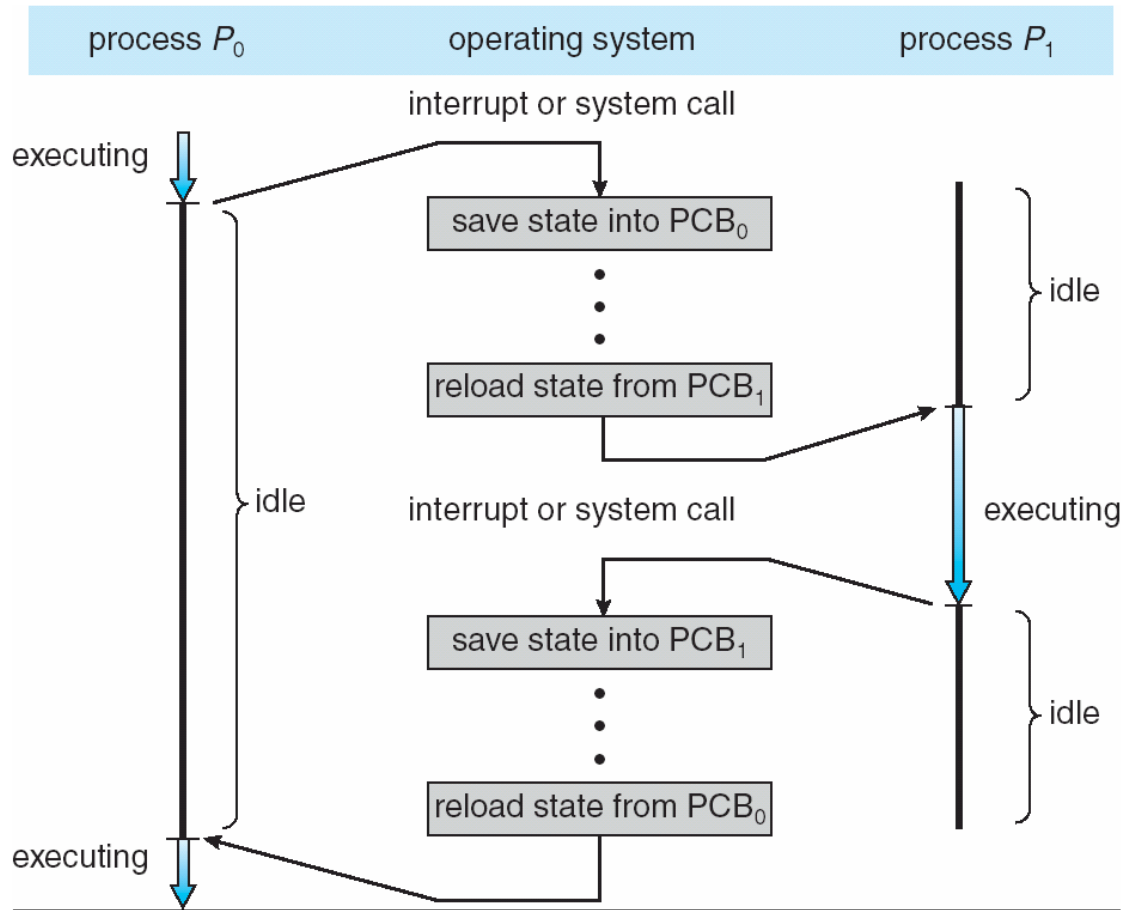
- Transition Running – Blocked:
 - Operating systems discovers that process cannot continue right now
 - Unix: process is automatically blocked by dispatcher
 - Other systems: processes may execute a system call such as “pause” to set themselves into blocked state
- Transition Running – Ready – Running:
 - Performed by dispatcher, without process knowing
 - When the time slice of a process expires, it is set into Ready state
 - Process is transferred back into Running state according to some scheduling policy (performance vs fairness)
- Transition Blocked – Ready
 - Process is transferred into Ready state, when external event occurs

Process Management



- Implementation:
 - Using one “Ready” and one “Blocked” queue
 - Weakness:
 - When a particular event occurs, ALL processes waiting for this event have to be transferred from the “Blocked” queue to the “Ready” queue
 - Operating system has to look through all the entries in the Blocked queue to select the right processes for transfer

Context Switch



Multithreading

- Processes have at least one thread of control
 - Is the CPU context, when process is dispatched for execution
- Multithreading is the ability of an operating system to support multiple threads of execution within a single process
- Multiple threads run in the same address space, share the same memory areas
 - The creation of a thread only creates a new thread control structure, not a separate process image

Threads

- The **unit of dispatching** in modern operating systems is usually a thread
 - Represent a single thread of execution within a process
 - Operating system can manage multiple threads of execution within a process
 - The thread is provided with its own register context and stack space
 - It is placed in the ready queue
 - Threads are also called “lightweight processes”
- The **unit of resource ownership** is the process
 - When a new process is created, at least one thread within the process is created as well
 - Threads share resources owned by a process (code, data, file handles)

Threads

- All threads share the same address space
 - Share global variables
- All threads share the same open files, child processes, signals, etc.
- There is no protection between threads
 - As they share the same address space they may overwrite each others data
- As a process is owned by one user, all threads are owned by one user

Threads vs Processes

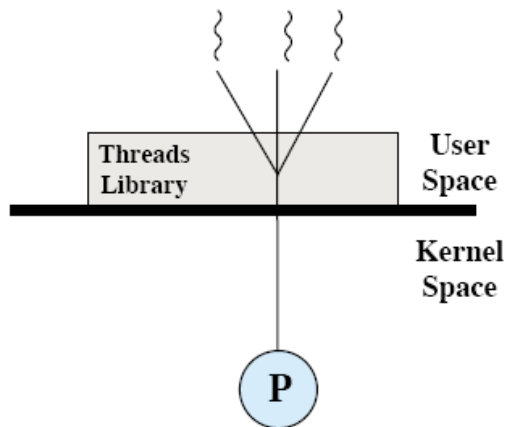
- Less time to create a new thread (10-times faster than process creation in Unix)
- Less time to terminate a thread
- Less time to switch between threads
- Threads enhance efficiency in communication between different communicating programs, no need to call kernel routines, as all threads live in same process context

Threads vs Processes

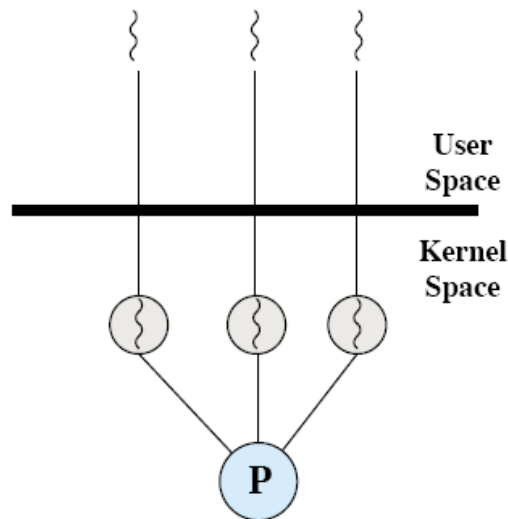
- Advantages of Threads
 - Much faster to create a thread than a process
 - Spawning a new thread only involves allocating a new stack and a new thread control block
 - Much faster to switch between threads than to switch between processes
 - Threads share data easily
- Disadvantages
 - Processes are more flexible
 - They don't have to run on the same processor
 - No protection between threads
 - Share same memory, may interfere with each other
 - If threads are implemented as user threads instead of kernel threads
 - If one thread blocks, all threads in process block

Thread Implementation

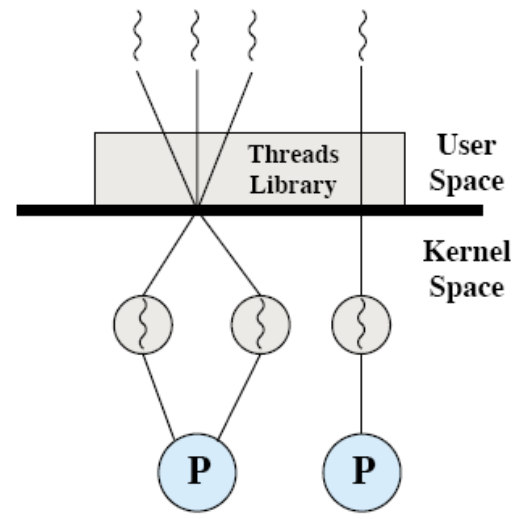
- Two main categories of thread implementation
 - User-level Threads (ULTs)
 - Kernel-level Threads (KLTs)
- Characterised by the extent of the kernel being involved in their management



Pure User-Level
ULT



Pure Kernel-Level
KLT



CombinedLevel
ULT/KLT

Scheduling

CPU Scheduler

- Scheduling can be preemptive or non-preemptive
- Non-preemptive:
 - once a process is scheduled, it continues to execute on the CPU, until
 - it is finished (terminates)
 - It releases the CPU voluntarily (cooperative scheduling behaviour, Ready queue)
 - It blocks due to I/O interrupts or because it waits for another process
 - No scheduling decision is made during clock interrupts, process is resumed after this interruption (unless a process with higher priority becomes ready)
- Preemptive:
 - A scheduled process executes, until its time slice is used up
 - Clock interrupt returns control of CPU back to scheduler at end of time slice
 - Current process is suspended and placed in the Ready queue
 - New process is selected from Ready queue and executed on CPU
 - Used by most operating systems

Scheduling Performance Criteria

- CPU utilisation
 - Keep the CPU as busy as possible
- Throughput
 - Number of processes that complete their execution per time unit
- Response time
 - The time it takes from when a request was submitted until the first response is produced by the operating system (latency)
- Turnaround time
 - Total amount of time to execute one process to its completion
- Waiting time
 - Total amount of time a process has been waiting in the Ready queue

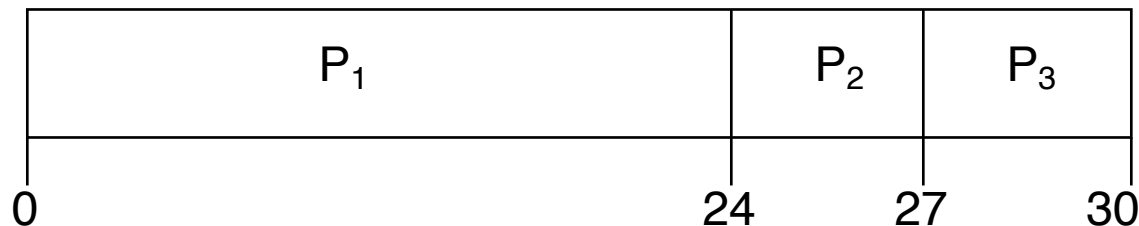
Optimisation Criteria

- Maximise CPU utilisation
- Maximise throughput
- Minimise turnaround time
- Minimise waiting time
- Minimise response time

First Come First Serve Scheduling (FCFS)

- Example:
 - Three processes arrive at the same time ($t = 0$)
 - They have different CPU burst times
- We assume the arrival order **P1, P2, P3**

Process	Burst Time
P1	24
P2	3
P3	3



- Waiting times
 - P1 has 0 time units waiting time
 - P2 has 24 time units waiting time
 - P3 has 27 time units waiting time
 - Average waiting time: $(0 + 24 + 27)/3 = 17$

First Come First Serve Scheduling (FCFS)

- Favours long processes
- CPU-intensive processes (long burst times)
- Disadvantages
 - Leads to poor utilisation of CPU and I/O devices
 - Average waiting time is highly variable
 - Short jobs may wait behind large ones

Round Robin (RR, Time Slicing)

- Is a preemptive scheduling policy
- Time quantum
 - Each process gets a fixed unit of CPU (usually 10-100 ms)
 - After time quantum has elapsed, the process is preempted and added to the end of the Ready queue
 - Processes scheduled in a cyclical fashion (always same sequence of processes) – round robin
- Fairness: Each process gets an equal time quantum
 - If the time quantum is q time units and if there are n processes in the Ready queue, each process get $1/n$ of the CPU time, in chunks with a size of at most q time units
 - No process waits more than $(n-1)q$ time units

Lottery Scheduling

- Basic idea
 - Give processes “lottery tickets” for various resources, such as CPU time
 - For each scheduling decision, a lottery ticket is chosen at random and the process holding that ticket gets resource
 - E.g.: CPU scheduling
 - Scheduler holds lottery 50 times per second
 - Each winner receives 20msec of CPU time

Scheduling in Real-Time Systems

- In real-time scheduling, time is essential
 - A process has to meet deadlines reliably
- A set of processes has to be scheduled so that all of them meet deadlines reliably
 - How to order / prioritise them?
 - How does the requirement of meeting deadlines and allow a system to reliably react to events impact on CPU utilisation?

Hard and Soft Real-Time

- Hard Real-Time
 - There are absolute deadlines that always have to be met
 - If a deadline is missed, there may be unacceptable danger, damage or a fatal error in the system controlled by real-time processing
- Soft Real-Time
 - Each processing of an event has an associated deadline that is desirable, but not mandatory
 - System continues to operate, even if a deadline has not been met
 - Missing an occasional deadline for scheduling a process (e.g. missing sensor input) is tolerable, system is able to recover from that

Effectiveness of a Periodic Scheduling Algorithm

- A periodic scheduling algorithm is effective when it guarantees that all hard deadlines are met
 - In order to meet all deadlines, schedulability must be guaranteed

Example Schedulability

- Three video streams handled by three processes
 - Stream 1 handled by process A:
 - Frame rate: Every 30 msec one frame
 - CPU time needed to decode stream: 10 msec per frame
 - Stream 2 handled by process B:
 - Frame rate: Every 40 msec one frame
 - CPU time: 15 msec
 - Stream 3 handled by process C:
 - Frame rate: Every 50msec one frame
 - CPU time: 5msec
- Each CPU burst handles one frame, must meet deadline (finished before next frame has to be displayed)
- Is our system performance good enough to schedule all three streams without delay?
- C_i / P_i is the fraction of CPU time used by process i
$$\frac{10}{30} + \frac{15}{40} + \frac{5}{50} = 0.8$$
 - E.g.: Process A consumes 10/30 of the CPU time available

Synchronisation

Resource Competition

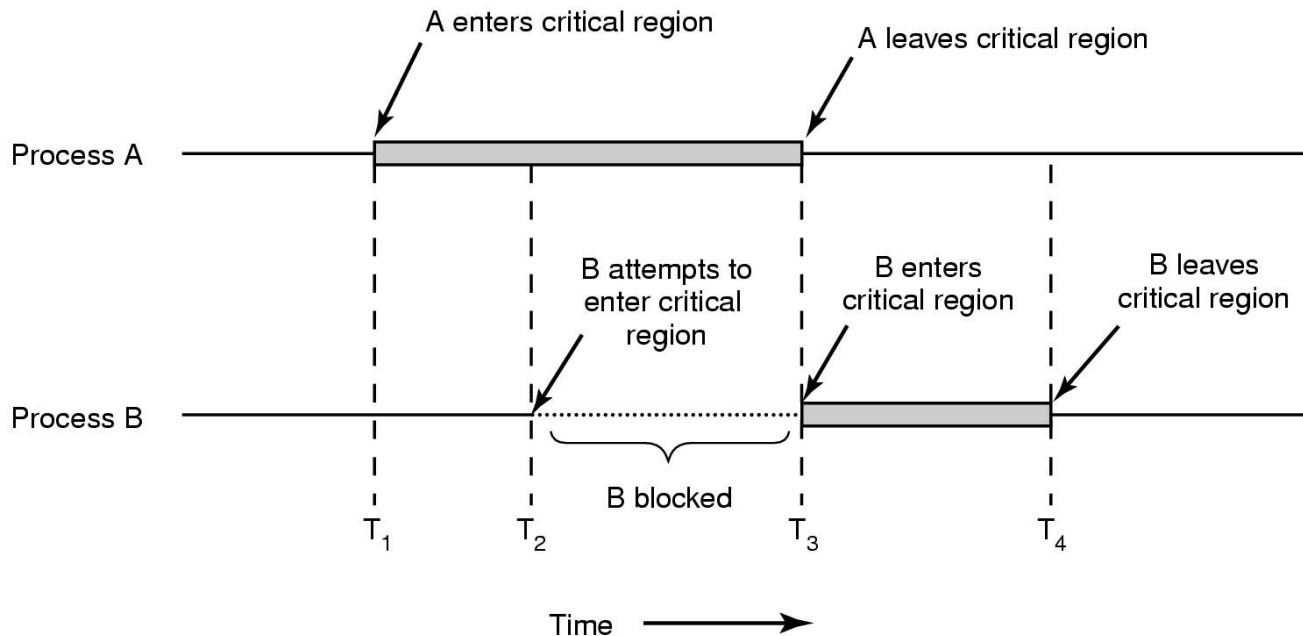
- Concurrent processes come into conflict when they are competing for use of the same resource
 - Beside shared data, this can also be I/O devices, processor time, system clock etc.
- Basic requirement
 - Enforce ***mutual exclusion*** between processes
 - Allow one process the execution of a course of actions, while excluding all other processes from performing this course of actions

Race Condition

- Occurs when multiple processes / threads read and write shared data items
- The processes race to perform their read / write actions
- The final result depends on the order of execution
 - The “winner” of the race is the process that performs the last update and determines the final value of a shared data item

The Critical Section Problem

- Avoid ***race conditions***
 - enforce ***mutual exclusion*** between processes
 - Strict serialisation
- Avoid ***deadlock and starvation***:
 - Enforcing mutual exclusion may result in deadlocks and starvation – has to be solved



The Critical Section Problem

- Avoid race conditions by enforcing ***mutual exclusion*** between processes
- Control entry to and exit from critical section
 - We need a Critical Section Protocol:
 - Entry section: Each process must request permission for entering a critical section
 - Requires Inter-process communication
 - has to wait / is suspended until entry is granted
 - Exit section:
 - Requires inter-process communication
 - process communicates that it leaves critical section
- Avoid deadlock and starvation:
 - Enforcing mutual exclusion may result in deadlocks and starvation – has to be solved

Requirements for Mutual Exclusion

- Serialisation of access:
 - Only one process at a time is allowed in the critical section for a resource
- A process that halts in its noncritical section must do so without interfering with other processes
- No deadlock or starvation
- A process must not be delayed in accessing a critical section
- A process remains inside its critical section for a finite time only

Solution to Critical Section Problem

- Modern solutions use a “lock”
 - Any solution to the critical section problem requires that critical sections are protected by some form of a “lock”
- Lock
 - A shared data item
 - Processes have to “acquire” such a lock before entering a critical section
 - Processes have to “release” a lock when exiting critical section

```
process ()  
{  
    acquire lock  
  
    critical_section() ;  
  
    release lock  
  
    remainder_section() ;  
}
```

Semaphore

```
process ()  
{
```


```
    wait(S)
```

```
    critical_section() ;
```


```
    signal(S)
```

```
    remainder_section() ;
```

```
}
```



```
wait(S) {  
    while(S <= 0) /* do nothing */;  
    S -- ;  
}
```

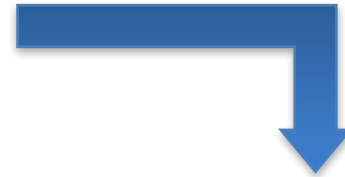


```
signal(S) {  
    S ++ ;  
}
```

- Semaphores are based on a decrement / increment mechanism:
 - Initialisation:
 - initialised with a non-negative integer value
 - Wait:
 - If semaphore $S \leq 0$, then process has to wait for signal
 - If semaphore $S > 0$, then process can proceed, S is **decremented**
 - Signal:
 - S is **incremented**

Counting also the Waiting Processes

```
wait(S) {  
    while(S <= 0) /* wait */;  
  
    // when wait is over decrement S  
    S -- ;  
}
```



```
wait(S) {  
    S -- ;  
    if(S < 0) /* wait in  
                blocked queue */;  
}
```

- All processes calling wait(S) first decrement the semaphore, then check whether to wait
- A semaphore showing a negative number indicates, that processes are blocked/waiting and how many processes are waiting

Counting also the Waiting Processes

```
process ()  
{
```


```
    wait(S)
```

```
    critical_section() ;
```


```
    signal(S)
```

```
    remainder_section() ;
```

```
}
```



```
wait(S) {  
    S -- ;  
    if (S < 0) /* wait in blocked  
                queue */;  
}
```



```
signal(S) {  
    S ++ ;  
    if (S <= 0) /* wakeup  
                  process */  
}
```

- For counting waiting processes, we need an **implementation** that suspends waiting processes and adds them to the Blocked queue

Implementation

- A semaphore is a data structure that contains

- A counter
- A waiting queue

```
typedef struct {  
    int counter;  
    Queue plist ;  
} Semaphore ;
```

- Semaphore can only be accessed by two atomic functions
 - wait (S) : decrements semaphore counter
 - If a process calls wait(), the counter is decremented, if it is zero – semaphore blocks calling process
 - signal(S) : increments semaphore counter
 - processes calling signal(S) wake up other processes

Counting also the Waiting Processes

Change of wait() procedure

```
wait(Semaphore S) {  
    if(S.value > 0) S.value -- ;  
    else {  
        add this process to S.plist;  
        block();  
    }  
}
```



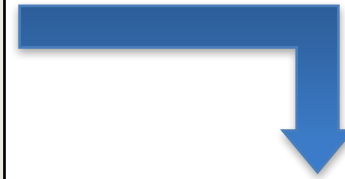
```
wait(semaphore S) {  
    S.value-- ;  
    if(S.value < 0) {  
        add this process to S.plist;  
        block();  
    }  
}
```

- If a process calls wait(S), it first decrements the semaphore, before it checks whether to wait
- A semaphore set to a negative number indicates, that there are blocked / waiting processes and how many processes are waiting

Counting also the Waiting Processes

Change of signal() Procedure

```
signal(Semaphore S) {  
    if(S.plist is empty) S.value ++;  
    else {  
        remove a process P from S.plist;  
        wakeup(P);  
    }  
}
```



```
signal(semaphore S) {  
    S.value++;  
    if(S.value <= 0) {  
        remove a process P from S.plist;  
        wakeup(P);  
    }  
}
```

- If a process calls signal(), it first increments the semaphore value
- A semaphore value lower or equal 0 indicates that there are processes waiting

Counting the Waiting Processes

Semaphore:

```
typedef struct {  
    int value ;  
    Queue plist ;  
} Semaphore ;
```

```
init(Semaphore S, int val) {  
    S.value = val;  
    S.plist = empty queue;  
}
```

```
process ()  
{
```

```
    wait(S)
```

```
        critical_section() ;
```

```
    signal(S)
```

```
        remainder_section() ;
```

```
}
```

- Semaphore counter:

- S.value ≥ 0
 - Number of available resources
- S.value < 0
 - The negative value expresses the number of processes waiting

```
wait(semaphore S) {  
    S.value-- ;  
    if(S.value < 0) {  
        add this process to S.plist;  
        block();  
    }  
}
```

```
signal(semaphore S) {  
    S.value++ ;  
    if(S.value  $\leq 0$ ) {  
        remove a process P from S.plist;  
        wakeup(P) ;  
    }  
}
```

Producer – Consumer Ring Buffer

```
init(mutex,1); init(full,0); init(empty, N);  
in = 0; out = 0; buffer[N] ;
```

Producer

```
while (TRUE)  
{  
    x = produce();  
    wait(empty);  
    wait(mutex);  
    append(x);  
    signal(mutex);  
    signal(full);  
}
```

Consumer

```
while (TRUE)  
{  
    wait(full);  
    wait(mutex);  
    y = take();  
    signal(mutex);  
    signal(empty);  
    consume(y);  
}
```

```
append(x)  
{  
    b[in] = x;  
    in = (in+1) mod N;  
}
```

```
take()  
{  
    y = b[out];  
    out = (out+1) mod N;  
    return y;  
}
```

- Bounded buffer:
 - Buffer limited to N places, is managed as a circular buffer

Dining Philosophers Problem

- Solution
 - 5 philosophers try to eat, allow only 4 philosophers at the table
 - at least one philosopher has access to two forks at a time,
- Only 4 processes at a time are allowed to execute “eat()”;
- Semaphores
 - Use a counting semaphore “seated” set to 4
 - One semaphore per fork
- No deadlock, no starvation

```
Semaphore fork[5] = {1,1,1,1,1};  
init(seated,4);
```

Process i

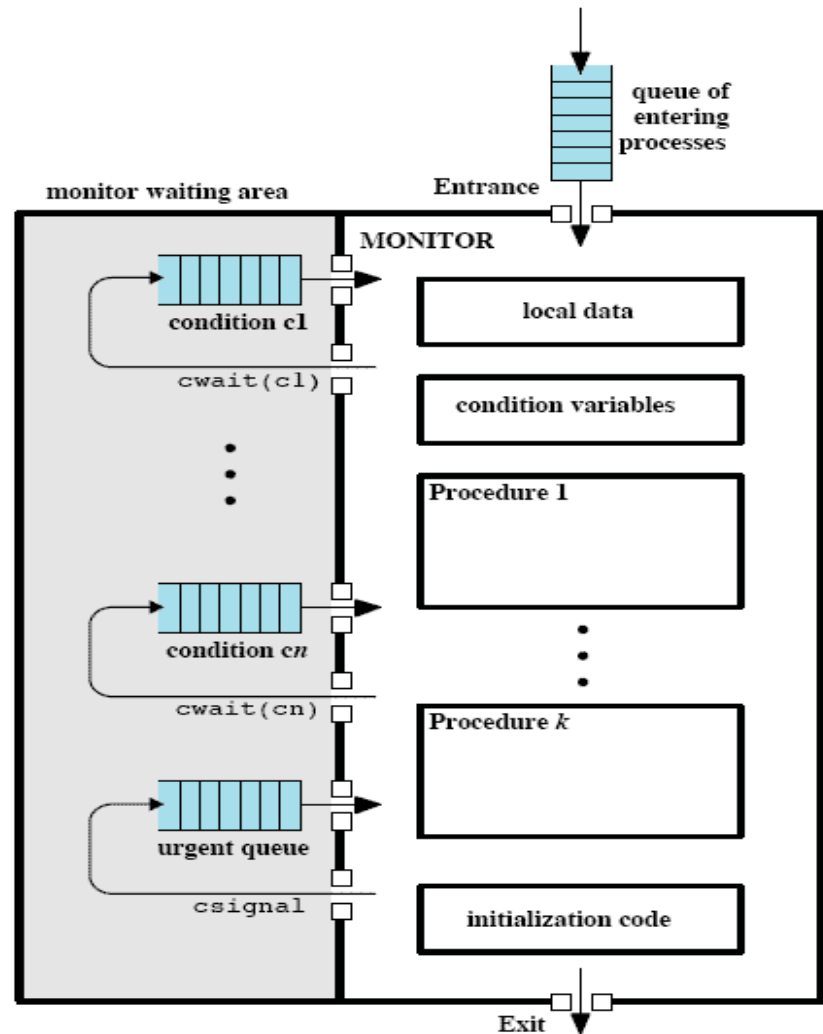
```
while (TRUE)  
{  
    think();  
    wait(seated);  
    wait(fork[i]);  
    wait(fork[(i+1) mod 5]);  
    eat();  
    signal(fork[(i+1) mod 5]);  
    signal(fork[i]);  
    signal(seated);  
}
```

Monitor

- A monitor is a software construct that serves two purposes:
 - enforces mutual exclusion of concurrent access to shared data objects
 - Processes have to acquire a lock to access such a shared resource
 - Support conditional synchronisation between processes accessing shared data
 - Multiple processes may use monitor-specific `wait()/signal()` mechanisms to wait for particular conditions to hold

Monitor

- Process “enters” a monitor
 - It calls one of the procedures
 - Process acquires lock, may have to wait
- One process currently in the monitor
 - Process may call *wait(condition)*
 - made to wait for a condition in a condition queue
 - Process may call *signal(condition)*
 - This resumes one of the processes waiting for this conditional signal



Producer – Consumer Ringbuffer Monitor

RingBuffer

Producer

```
while(TRUE)
{
    x = produce();
    append(x);
}
```

Consumer

```
while(TRUE)
{
    y = take();
    consume(y);
}
```

```
in = 0;
out = 0;
count = 0;
buffer[N];
notfull;
notempty;
```

```
void append(char item) {
    if (count == N) wait(notfull);
    b[in] = item;
    in = (in+1) mod N;
    count++;
    signal(notempty);
}
```

```
char take() {
    if (count == 0) wait(notempty);
    item = b[out];
    out = (out+1) mod N;
    count--;
    signal(notfull);
    return item;
}
```

Memory Management

Memory Abstraction: Address Space

- Programs operate with an abstraction of physical memory
 - Is a contiguous sequence of “logical” memory addresses
 - Each process lives within its own address space
 - Logical addresses are translated into actual physical memory addresses, whenever processes access memory locations via logical addresses
- Supports
 - Relocation
 - Protection

Memory Abstraction: Address Space

- Address space
 - Is a contiguous sequence of “logical” memory addresses
 - Each process lives within its own address space
 - Size of address space depends on processor architecture
 - 32bit architecture: 4 GB, 4 billion logical addresses

Bytes	Exponent				
1,024	2^{10}	1kb	1024bytes		
1,048,576	2^{20}	1MB	1024kb		1024 x 1024
1,073,741,824	2^{30}	1GB	1024MB		1024 x 1024 x 1024
4,294,967,296	2^{32}	4GB	4 x 1024MB		4 x 1024 x 1024 x 1024
1,099,511,627,776	2^{40}	1TB	1024GB		1024 x 1024 x 1024 x 1024
1,125,899,906,842,620	2^{50}	1PB	1024TB		1024 x 1024 x 1024 x 1024 x 1024
1,152,921,504,606,850,000	2^{60}	1EB	1024PB		1024 x 1024 x 1024 x 1024 x 1024 x 1024
18,446,744,073,709,600,000	2^{64}	16EB			16 x 1024 x 1024 x 1024 x 1024 x 1024 x 1024

Internal Fragmentation

- Programs may not fit into a partition
- Inefficient use of main memory
 - A program will occupy a complete partition, even if it is small and does not need the complete memory allocated
 - Waste of memory: a partition is *internally* fragmented, free memory fragments of a partition cannot be given to other processes
- This phenomenon is called ***internal fragmentation***
 - the block of data loaded (swapped in) is smaller than the partition

External Fragmentation

- The memory “holes” between assigned partitions may be too small for any process to fit in – waste of memory
- This phenomenon is called ***external fragmentation***
 - Memory external to partitions becomes increasingly fragmented
 - Memory utilisation declines

Virtual Memory Management

- Modern approach
 - All memory references are logical addresses that are dynamically translated into physical addresses at run time
 - Non-contiguous allocation of memory for processes
 - A process may be broken up into a number of pieces – pages and/or segments that don't need to occupy a contiguous area of main memory
 - Processes only partially loaded into physical memory

Paging

- Paging allows non-contiguous allocation of memory
 - Allocation of physical memory for a process image that is non-contiguous
- Break up both virtual address space of process image and physical memory into fixed-sized blocks:
 - **Frames**: Physical memory is divided into fixed-sized blocks, called “frames” or “page frames”
 - **Pages**: Virtual address space of a process is divided into fixed-sized blocks called “pages”
 - Any page can be assigned to any free page frame

Paging

- Paging avoids external fragmentation and the need for compaction
- Helps to implement virtual memory
 - Page file on hard disk, is also organised in blocks of the same size as a page
 - Pages of a process can easily be swapped in and out of memory
 - We can implement a process image only being partially loaded, pages are loaded on demand
 - No time-consuming search for free best-fit memory fragments, no external memory fragmentation
- Paging is supported by modern hardware

Addressing Virtual Memory

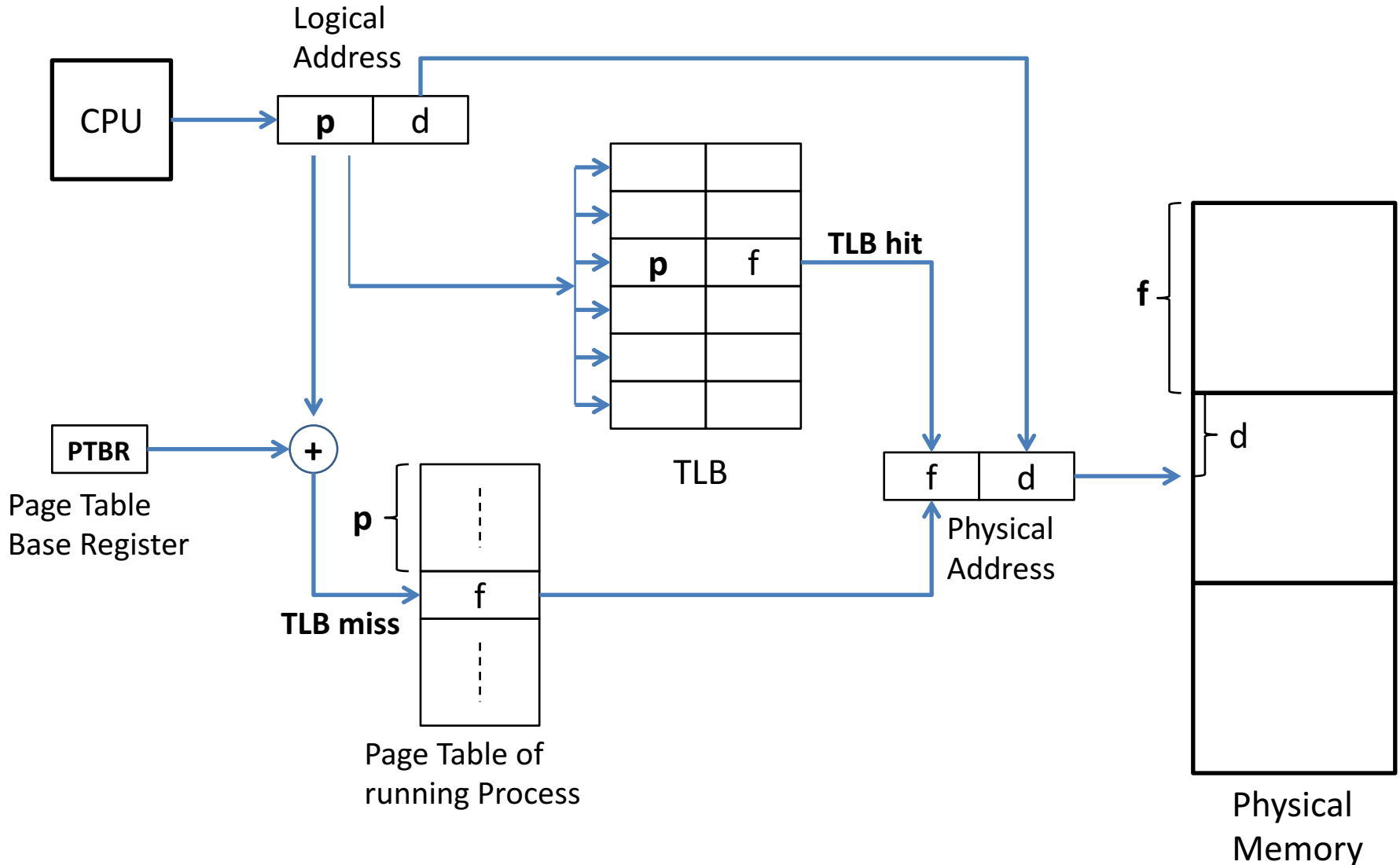
- Pages are typically of sizes between 512 bytes and 16MB
- Consider a 32-bit memory address
 - 4kB per page: we need 12 bits ($2^{12} = 4096$) to address a location within the page
 - 20 bits for the page number: we can have $2^{20} = 1$ Mio pages
 - **Page table must have 1 Mio entries**
- Address space: 4GB of virtual memory

Bytes	Exponent				
1,024	2^{10}	1kb	1024bytes		
1,048,576	2^{20}	1MB	1024kb		1024 x 1024
1,073,741,824	2^{30}	1GB	1024MB		1024 x 1024 x 1024
4,294,967,296	2^{32}	4GB	4 x 1024MB		4 x 1024 x 1024 x 1024
1,099,511,627,776	2^{40}	1TB	1024GB		1024 x 1024 x 1024 x 1024
17,592,186,044,416	2^{44}	16TB	16 x 1024GB		16 x 1024 x 1024 x 1024 x 1024
1,125,899,906,842,620	2^{50}	1PB	1024TB		1024 x 1024 x 1024 x 1024 x 1024
1,152,921,504,606,850,000	2^{60}	1EB	1024PB		1024 x 1024 x 1024 x 1024 x 1024 x 1024
18,446,744,073,709,600,000	2^{64}	16EB			16 x 1024 x 1024 x 1024 x 1024 x 1024 x 1024

Translation Lookaside Buffer (TLB)

- Cache for a subset of page table entries (the most used pages)
- Using the TLB
 - CPU generates logical address
 - “TLB hit”: If lookup of page number in TLB is successful:
 - the start address of the frame is immediately available for memory access
 - “TLB miss”: If lookup of page number in TLB is not successful:
 - the page table in memory has to be accessed first to retrieve the start address of the frame
 - Page number and found frame are added to the TLB (potentially replacing other entries according to a replacement policy)
- TLB must be loaded with process-specific page table entries at each context switch

Translation Lookaside Buffer (TLB)



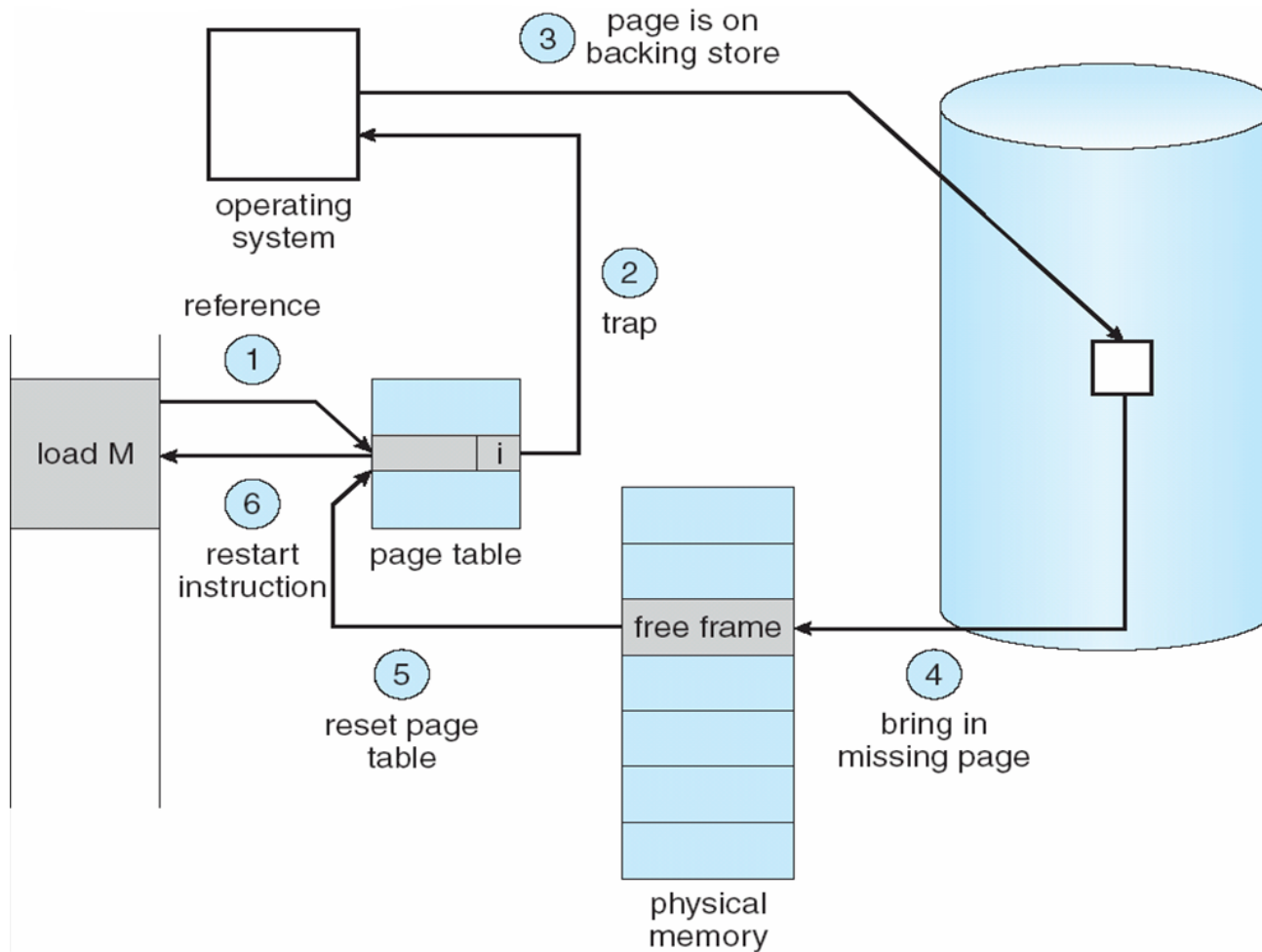
Fragmentation

- There is no external fragmentation
 - Any free frame can be allocated to a process that needs it
 - This is due to the fixed page size the correspondence between page size and frame size
- There is internal fragmentation
 - As page / frame sizes are fixed, there may be some waste
 - The last frame allocated to a process may not be needed in its completeness
 - Worst case: process may need n frames plus one additional byte! A complete additional frame has to be allocated just for this additional byte of required memory
 - Expected internal fragmentation: on average one-half page per process

Page Table Management

- Size of address space of a process influences the maximum size of the page table
 - Can be huge for address spaces 2^{32} (4GByte) or 2^{64} (16 Exabytes!)
 - Example:
 - Address space: 32-bit logical address space, 4GB
 - Page size: 4kb (2^{12}), address space is composed of 2^{20} ($2^{32} / 2^{12}$) pages (1 Mio)
 - page table may contain up to 1 Mio entries (as many as the process needs)
 - If each entry in page table is 4 bytes (32-bit), then we may need up to 4MB of physical memory to hold this page table
- We may not be able to allocate a contiguous area of physical memory
- The page table itself may be subject to paging

Page Fault Handling



Resident Set

- Is the number of pages currently loaded into physical memory
- Is restricted by the number of frames a process may maximally use

Principle of Locality

- References to memory locations within a program tend to cluster
 - if there is an access to a memory location, the next access is, in all likelihood, very near to the previous one
 - Loading one page may satisfy subsequent memory access, as all required memory locations may be contained within this page
- We can observe that only a few pieces of a process image will be needed during short time periods
 - This is the current memory “locality” of a process
 - It is therefore possible to make informed guesses about which pieces will be needed in the near future
- This principle indicates that virtual memory management can be managed efficiently

Replacement Policy

- Deals with the selection of a page in main memory to be replaced when a new page must be brought in
- Which page should be replaced?
 - Page removed should be the page least likely to be referenced in the near future
- How is that determined?
 - Most policies predict the future behaviour on the basis of past behaviour
- Metrics
 - Minimize page-fault rate
 - Avoid replacement of pages that are needed for the next memory references

Page Replacement Algorithms

- The optimal policy (OPT) as a benchmark
- Algorithms used
 - Least recently used (LRU)
 - First-In-First-Out (FIFO)
 - Clock Algorithm

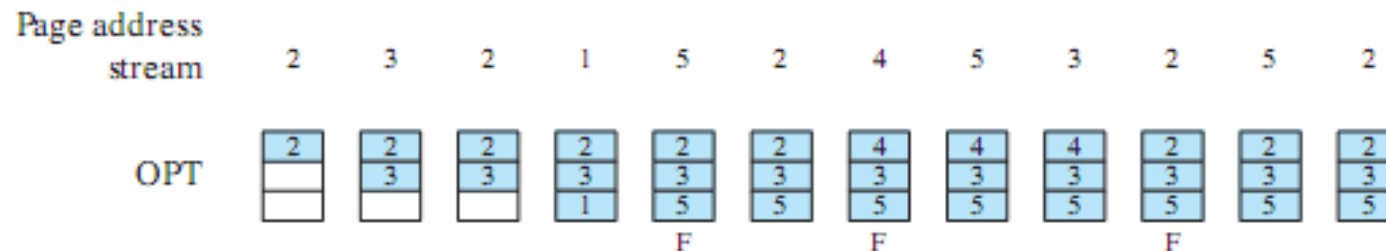
Optimal Policy OPT

- The theoretically optimal replacement policy
 - Look into the future: select the page for which the time until it is referenced again is the longest among all other pages
- Is not a realistic strategy
 - This policy is impossible to implement, as it would require an operating system to have perfect knowledge about all future events
- But we can use it as a **benchmark**
 - Gives us the minimum number of page faults possible
- How to do that
 - Record a finite sequence of page references
 - Replay this sequence: in hindsight, we know now which page can be replaced

Optimal Policy OPT

- Assumption
 - 3 memory frames available
 - Process has 5 pages
 - When process is executed, the following sequence of page references occurs (called a page reference string):

Page reference string: 2 3 2 1 5 2 4 5 3 2 5 2





F = page fault occurring after the frame allocation is initially filled

OPT policy results in 3 replacement page faults (minus the initial ones to fill the empty frames, total page faults are 6)

Optimal Policy OPT

Reference to page 2:
- **We replace page 4 in
Frame 1**



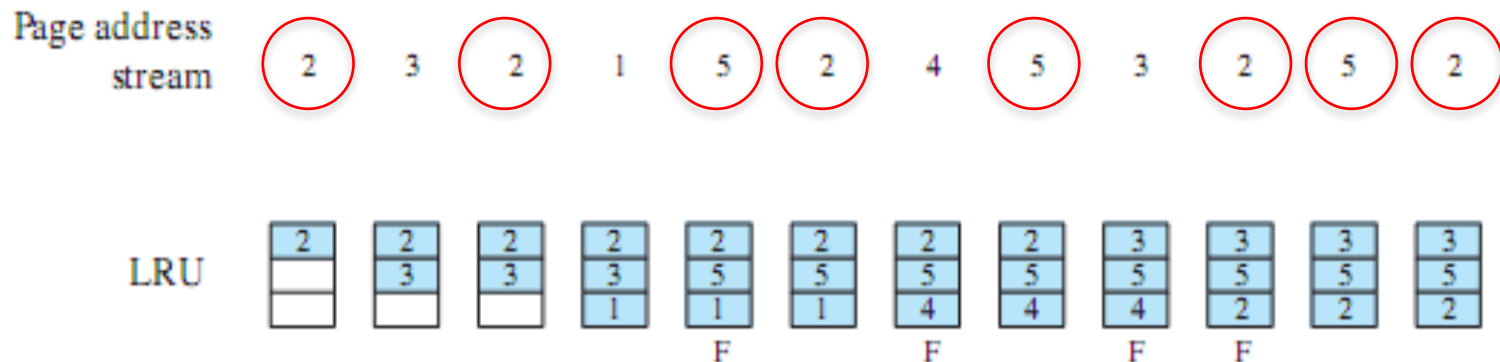
Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2	2	2	2	4	4	4	2	2	2
Frame2		3	3	3	3	3	3	3	3	3	3	3
Frame3				1	5	5	5	5	5	5	5	5
Page fault					F		F			F		

- In this particular example, the “optimal” policy would create 3 replacement page faults
- In reality, we cannot know the future, therefore we may have more page faults
- OPT is our “benchmark”

Least Recently Used (LRU)

- Idea: past experience gives us a guess of future behaviour
- Replaces “least recently used” page: replace the page that has not been referenced for the longest time
 - “Least recently used” page should be the page least likely to be referenced in the near future
- Pages that are more frequently used remain in memory
- Good behaviour in terms of page faults
 - Almost as good as OPT
- Difficult to implement
 - Search for oldest page may be costly
 - One approach would be to tag each page with the time of the last reference
 - Requires great deal of overhead to manage

LRU Policy




F = page fault occurring after the frame allocation is initially filled

- The LRU policy does almost as well as the theoretical optimal policy OPT
 - LRU recognises that pages 2 and 5 are referenced more frequently than other pages, whereas FIFO does not
- We have to look into the past to guess what trend future page references may follow

LRU Policy



Reference to pages 4:
- **We replace page 1 in Frame 3**

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2	2	2	2	2					
Frame2		3	3	3	5	5	5					
Frame3				1	1	1	4					
Page fault					F		F					

Reference to page 3:
- **We replace page 2 in Frame 1**

Reference to page 2:
- **We replace page 4 in Frame 3**

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2	2	2	2	2	2	3	3	3	3
Frame2		3	3	3	5	5	5	5	5	5	5	5
Frame3				1	1	1	4	4	4	2	2	2
Page fault					F		F		F	F		

- With LRU, we replace page 2 with page 3, and immediately afterwards, we need page 2 again
- LRU is not able to detect this situation
- However, it is a strategy that comes close to OPT

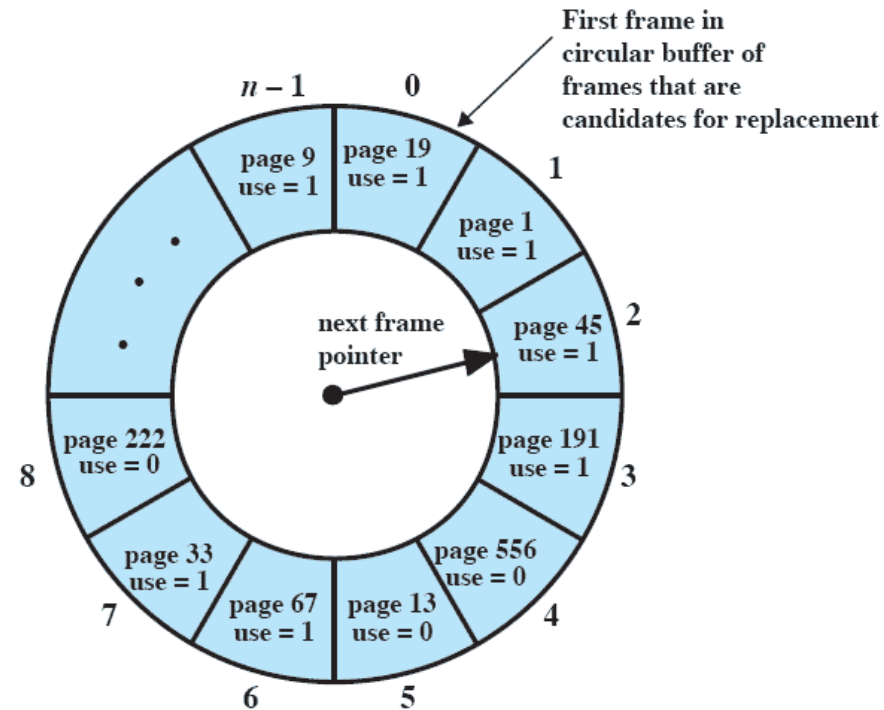
First-In-First-Out (FIFO)

- Treats memory frames allocated to a process as a circular buffer
- Pages are replaced in a round-robin style
 - Is the simplest replacement policy to implement
- When a page has to be replaced, then the “oldest” page will be chosen
 - What if this page is immediately needed again?
- Problem
 - Does not indicate how heavily a page is actually used

Clock Policy

Implementation of Second-Chance Algorithm

- The set of frames is considered as laid out like a circular buffer
 - contains a FIFO list of pages
 - Can be circulated in a round-robin fashion
- Each page has a reference bit
 - When a page is first referenced (a page fault that leads to its load), the use bit of the frame is set to 1
- Each subsequent reference to this page again sets this bit to 1



(a) State of buffer just prior to a page replacement

Clock Algorithm

Implementation of Second-Chance Algorithm

- Initially, all frames in circular buffer are empty
 - Each frame entry has a use bit (reference bit)
- Frame pointer: is regarded the “clock hand”, is called the “next frame” pointer
- Procedure
 - Occurrence of page fault:
 - Progress position pointer to first unused frame (use bit == 0)
 - During this progression: set use bit of visited frame entries to 0
 - Load page into free frame, set use bit to 1
 - Move position pointer to next frame
- When the circular buffer is filled with pages, the “next frame” pointer has made one full circle
 - it will be back at the first page loaded
 - points to the “oldest” page

Thrashing

- Thrashing is a situation where a process is repeatedly page faulting
 - The process does not have enough frames to support the set of pages needed for fully executing an instruction
 - It may have to replace a page that itself or another process may need again immediately
- A process is thrashing if it is spending more time on paging than execution

Thrashing

- Important: need of clever page replacement algorithms
 - Don't replace a page you may need in the immediate future
 - Avoid unnecessary writes for unchanged pages
- Thrashing
 - A state in which the system spends most of its time swapping pieces rather than executing instructions
- Operating tries to avoid thrashing by guessing which loaded pages are least likely to be used in the near future – candidates for being replaced

Prevent Thrashing

- If we want to prevent thrashing, we must provide a process with as many frames as it “needs”
- How many frames are needed?
- Principle of Locality
 - A “locality” is a set of pages that are **actively** used together during program execution
 - A program is generally composed of several different localities that may overlap
 - As a process executes, it moves from locality to locality

Frame Allocation to Processes

- Each process is allocated free memory
 - Each process has a Free-frame list
 - Demand-paging will allocate frames from this list due to page faults
 - When the free-frame list is exhausted, a page replacement algorithm will choose a page to be replaced
- With multiple processes: how many free frames should be allocated to each process?

Minimum Number of Frames

- Performance depends on a minimum number of frames allocated
 - If number of free frames decreases, page faults increase
 - Increase of page faults slows process execution
 - I/O operations
 - Instruction that leads to a page fault has to be restarted (effectively executing instructions twice)
- We need enough frames to hold all the pages in memory that are referenced by any single instruction – principle of locality
- We have to manage the size of the resident set

Resident Set

- Is the number of pages currently loaded into physical memory
- Is restricted by the number of frames a process may maximally use

Working Set

- Working Set

Set of pages a process is currently working with and which should be resident in memory to avoid thrashing

- A working set is an approximation of a program's locality
- It is the set of pages actively used by a process and all these pages should be held in memory
- The size of this set gives us an estimate how many frames should be allocated to a process – how large the resident should be

Working Set

- Provides insight in the required amount of page frames for a process
- What to do if the Resident set is too small
 - Consult the “working set”: Add more frames
 - If we ran out of frames, suspend process and try to allocate again later
- Periodic discarding of pages from the resident set that are not in the working set
 - These are pages that are loaded into memory, but haven't been referenced for a long time

Working Set

- A working set is an approximation of a program's locality
- Uses the most recent page references from a program to determine which pages are most important
 - Parameter Δ : the most recent Δ page references
- Defines a “working set window”
 - The set of pages that are referenced within the recent Δ page references
 - A page in active use is part of the working set window
 - If a page is not used for Δ time units, it will be dropped from the working set window

Working Set Size WSS

- Parameter Δ : number of most recent page reference observations
 - If Δ is too small, not all actively used pages will be in the working set window
 - If Δ is too large, it may overlap several localities
- Working set size WSS_i
 - Size of the working set for process i
 - Depends on how many past observations Δ are taken into account
 - Process is actively using all the pages in its working set
- Therefore, a process needs at least WSS_i frames to avoid thrashing
- Operating system monitors the working set of a process and allocates WSS_i frames to the process

File Management

Files

- Files provide a way to store information on disk
- Files are an abstraction concept
 - Users operate with a simple model of a byte stream being written to or read from disk
 - Operating system hides details how disk space is allocated to store the information represented by a file

File System

- Provides an organised, efficient and secure access to data on secondary storage
- Organisation
 - Organises data into files, directories
 - Supports APIs to manipulate these management objects
- Efficiency
 - Perform I/O in blocks rather than bytes, random access to any block on a disk
- Security and integrity
 - Define ownership of files, permissions
 - Allow recovery from failures, avoid data loss

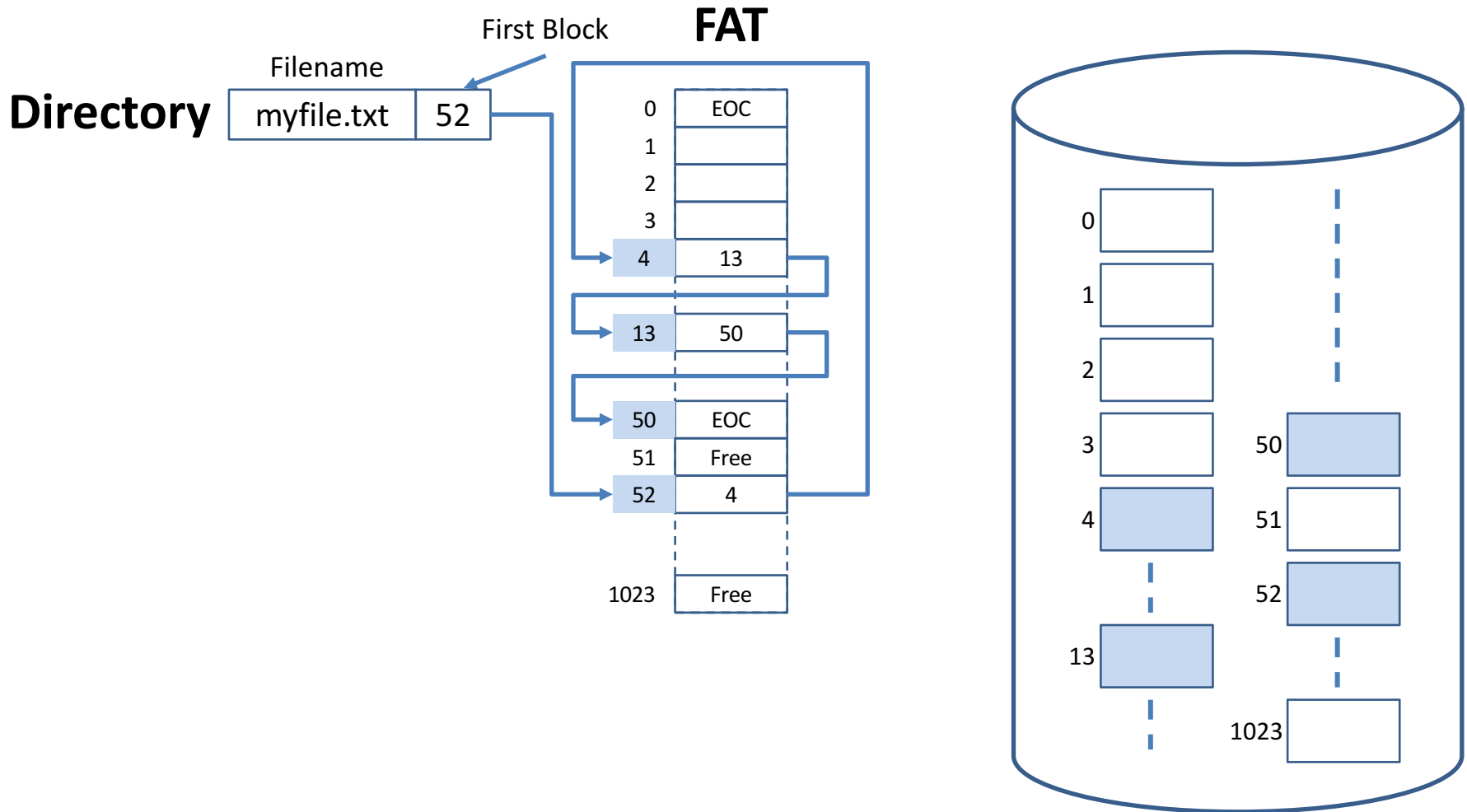
File Allocation Method

- Disks are organised in a block structure, each block of a particular size
- A file is stored on disk as a collection of these blocks
 - Blocks are allocated to files
- Block allocation strategies
 - Contiguous allocation
 - Non-contiguous allocation:
 - chained allocation
 - Indexed allocation
 - FAT, i-Nodes

File Allocation Methods

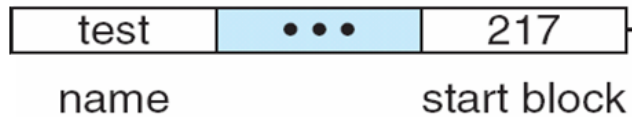
- File Allocation Table (FAT)
 - Is a linked-list allocation form
 - Eliminates disadvantages by taking the pointer of the block and holding it in an extra table – the File Allocation Table
 - File Allocation Table held in memory, used to find file
 - Actual blocks don't waste space for pointers
 - All pointers now in main memory, can easily be followed for find a block address and performing one I/O action to load block

File Allocation Table FAT

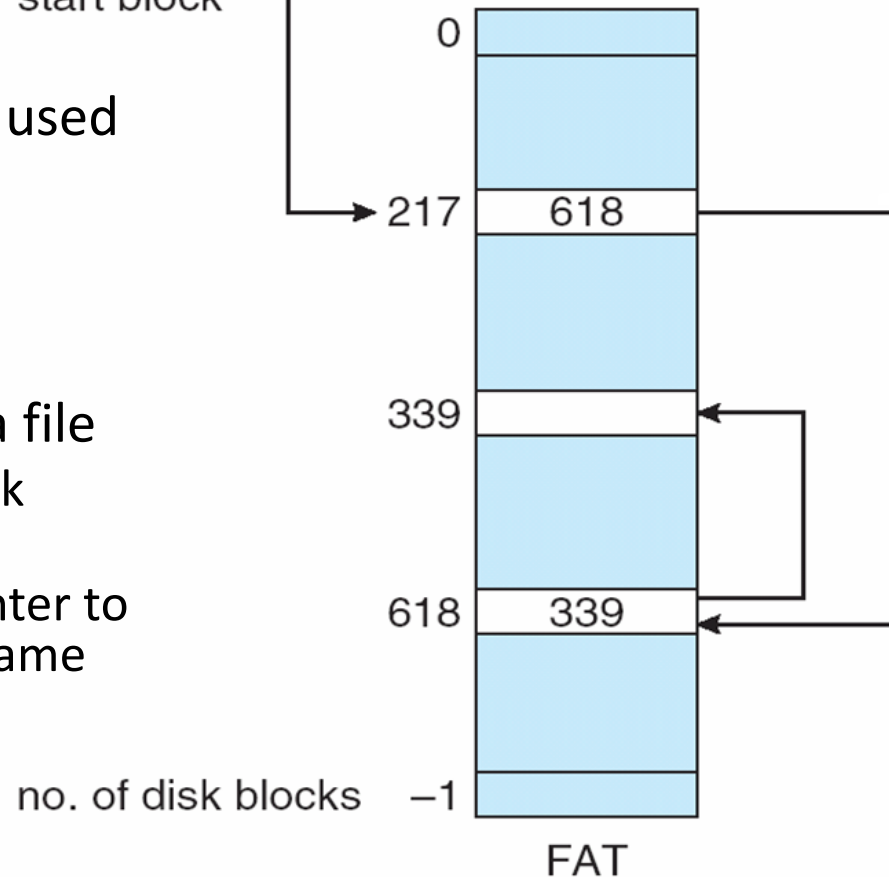


File Allocation Table FAT

directory entry



- File Allocation Table held in memory, used to find file block
 - Actual blocks don't waste space for pointers
 - All pointers now in main memory
- Entries in FAT form a block chain for a file
 - The index of the FAT entry is the block address of a file
 - The content of the FAT entry is a pointer to the next FAT entry of a block of the same file



Size of FAT

- Storage: 8GB USB drive, Block size: 4KB
- How many blocks do we need on the disk for the FAT?
- Remember:
 - $8\text{GB} = 8 \times 1024 \times 1024 \times 1024 \text{ bytes}$
 - $4\text{KB} = 4 \times 1024 \text{ bytes}$
- We calculate:
 - $8\text{GB} / 4\text{KB} = 8 \times 1024 \times 1024 \times 1024 \text{ bytes} / 4 \times 1024 \text{ bytes} = 2 \times 1024 \times 1024 = 2 \text{ Mio blocks}$
- Addressing:
 - We need at least 2^{21} entries in the FAT to address all 2 Mio blocks (2×2^{20})
 - We choose a 32-bit format for FAT entries (4 bytes), 1 block can hold 1024 entries: $4 \times 1024 \text{ bytes} / 4 \text{ bytes} = 1024 \text{ entries}$
- Space for FAT on disk
 - $2 \times 1024 \times 1024 \text{ entries} / 1024 \text{ entries} = 2 \times 1024 = \mathbf{2048 \text{ blocks for the FAT}}$

Bytes	Exponent			
1,024	2^{10}	1kb	1024bytes	
1,048,576	2^{20}	1MB	1024kb	1024 x 1024
1,073,741,824	2^{30}	1GB	1024MB	1024 x 1024 x 1024
4,294,967,296	2^{32}	4GB	4 x 1024MB	4 x 1024 x 1024 x 1024
1,099,511,627,776	2^{40}	1TB	1024GB	1024 x 1024 x 1024 x 1024
1,125,899,906,842,620	2^{50}	1PB	1024TB	1024 x 1024 x 1024 x 1024 x 1024
1,152,921,504,606,850,000	2^{60}	1EB	1024PB	1024 x 1024 x 1024 x 1024 x 1024 x 1024
18,446,744,073,709,600,000	2^{64}	16EB		16 x 1024 x 1024 x 1024 x 1024 x 1024 x 1024

Indexed Allocation

- Addresses problems of contiguous and linked-list allocation
 - Fast load of complete file
 - Random access to a particular block
- Indexes are hierarchical
 - One block can only contain a small list of addresses to disk blocks
 - Solution: multiple levels
 - Entry in index table points to another index table

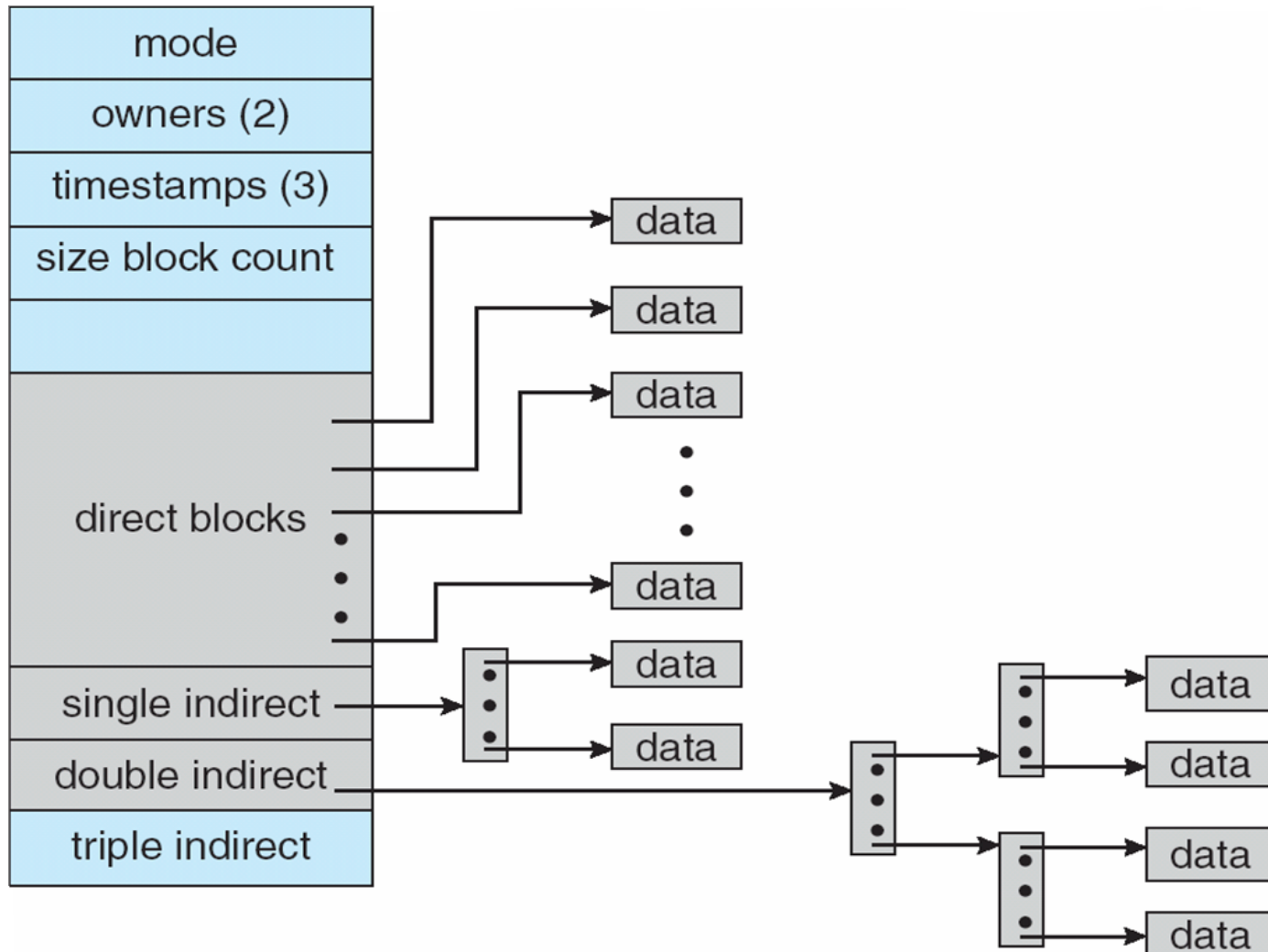
Inodes

- All types of Unix files are managed by the operating system by means of inodes
 - Is a control structure (index node) that contains the key information needed by the operating system for a particular file
 - Describes its attributes
 - Describes the layout of the file (allocated blocks)
- There can be several file names for a single inode
- But:
 - An active inode is associated with exactly one file
 - Each file is controlled by exactly one inode

File Allocation with Inodes

- Inode contains index referencing allocated blocks
 - First N entries point directly to the first N blocks allocated for the file
 - If file is longer than N blocks, more levels of indirection are used
 - Inode contains three index entries for “indirect” addressing
 - “single indirect” address:
 - Points to an intermediate block containing a list of pointers
 - “double indirect” address:
 - Points to two levels of intermediate pointer lists
 - “triple indirect” address:
 - Points to three levels of intermediate pointer lists
- The initial direct addresses and the three multi-level indirect addressing means form the index

Inode Indexed References of Disk Blocks



File Allocation with i-Nodes

- What is maximum size of a file that can be indexed:
 - Depends of the capacity of a fixed-sized block
- Example implementation with 15 index entries:
 - 12 direct, single (13) / double (14) / triple (15) indirect
 - Block size 4kb, holds 512 block addresses (32-bit addresses)

Level	Number of Blocks	Number of Bytes
Direct	12	48K
Single Indirect	512	2M
Double Indirect	$512 \times 512 = 256K$	1G
Triple Indirect	$512 \times 256K = 128M$	512G

Journaling File System

- Idea
 - Keep a small circular log on disk
 - Record in this what write actions will be performed on the file system, before actual disk operations occur
- Log file helps to recover from system crashes
 - After a system crash, operating system can take information from log file and perform a “roll forward”:
 - Finish write operations as recorded in journal

Journaling File System

- Manipulation done in the form of transactions
 - First, recording of operations in log (begin of transaction)
 - All three actions necessary for deleting a file are recorded
 - Log entry is written to disk
 - Second, after log recordings, actual write operations begin
 - Third, when all write operation on disk successful (across system crashes), log entry is deleted (end of transaction)

Journaling File Systems

- Robust in case of system crashes
 - Logs are checked after recovery and logged actions redone
 - this can be done multiple times if there are multiple system crashes
- Changes to file system are atomic
- Logged operations must be “idempotent”
 - Can be repeated as often without harm
 - E.g.: “mark i-node k as free” can be done over and over again

Deadlock

Deadlock Conditions

- There are 4 deadlock conditions that, when they all hold, will necessarily lead to a deadlock

(Three pre-conditions)

1. Mutual Exclusion

- There are mechanisms to enforce mutual exclusion between processes

2. Hold and Wait

- Process can hold resources, while it is waiting for other resources to become available

3. No Preemption

- Processes cannot be interrupted to free their resources by force

(Occurrence of a particular sequence of events)

4. Circular Wait

- Processes wait for each other to release resources they want to acquire, they form a kind of “closed chain of processes”

Handling Deadlocks

- Deadlock prevention
 - Avoid at least one of the 4 deadlock conditions
- Deadlock avoidance
 - The reservation of resources that would lead to deadlock, are not granted
- Deadlock detection
 - There is no restriction on resource allocation
 - There is a periodic check whether a deadlock is occurring
 - In case of a detected deadlock, recovery mechanisms are employed