Multiple Choice: (1 mark/ question)

- Dining phylosopher

Dining philosophers with semaphores

```
#define N 5 /* number of philosophers */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY 1 /* philosopher is trying to get forks */
#define EATING 2 /* philosopher is eating */

int state[N]; /* each philosophers is either THINKING, HUNGRY or EATING, initialized to THINKING */
sem_t cs_m; /* semaphore for the critical section, shared by all philosophers, initialized to 1 */
sem_t p_m[N]; /* one semaphore per philosopher, initialized to 0 */

int left(i) { return (i+N-1) % N; }
int right(i) { return (i+1) % N; }

void philosopher(int i) /* to be executed by different threads */
{
  while (1) {
    think( ); /* philosopher is thinking */
    take_forks(i); /* acquire two forks or block */
    eat( ); /* yum-yum, spaghetti */
    put_forks(i); /* put both forks back on table */
  }
}
```

```
void take_forks(int i) {
  down(& cs_m);           /* enter critical region */
    state[i] = HUNGRY;    /* record fact that philosopher i is hungry */
    test(i);              /* try to acquire 2 forks */
  up(& cs_m);             /* exit critical region */
  down(& p_m[i]);         /* block if forks were not acquired */
}

void put_forks(i) {       /* i: philosopher number, from 0 to N-1 */
  down(& cs_m);           /* enter critical region */
    state[i] = THINKING;  /* philosopher has finished eating */
    test(left(i));        /* see if left neighbor can now eat */
    test(right(i));       /* see if right neighbor can now eat */
  up(& cs_m);             /* exit critical region */
}

void test(i) {            /* i: philosopher number, from 0 to N-1 */
  if (state[i] == HUNGRY && state[left(i)] != EATING && state[right(i)] != EATING) {
    state[i] = EATING;    /* only start eating if hungry, and neighbors not eating */
    up(& p_m[i]);
  }
}
```

5 processes/threads, each needs two resources, such as files, to proceed how to allocate resources so that all process/threads get to execute?

Attempt 1: repeat forever: (grab left fork ; grab right fork; eat; put forks back; think)

this would lead to a **deadlock:** assuming all philosophers are reasonably synchronized----each philosopher will end up with one fork (left)--- each philosopher will block on trying to get the other fork (right)---nobody gets to eat at all

Attempt 2:

```
repeat forever:
    repeat:
        try to grab left fork
        try to grab right fork
        if both forks grabbed then break
        else put any grabbed forks back and take a short nap
    eat
    put forks back
    think
```

Attempt 3: same as before, but there is one pink hat--- only a philosopher with a pink hat will eat-- after eating, the philosopher passes the hat to another philosopher

yes it would, but…

- only one philosopher is eating at any given time
- but with 5 forks, 2 philosophers could be eating at the same time
- non-optimal use of resources, resulting in reduced parallelism
- called arbitrator solution

Attempt 4:

```
repeat forever:
    repeat:
        try to grab left fork
        try to grab right fork
        if both forks grabbed then break out of loop
        else put any grabbed forks back and take a short RANDOM nap
    eat
    put forks back
    think
```

random timeout mechanism for preventing deadlocks

- likely to work, often used in real world, for example in networking
- but… still a small chance for starvation:
  - if sleep is the same for all philosophers, no one gets to eat
  - in some cases some philosophers might never get to eat --or some philosophers will get to eat less often than others (fairness problem)

Attempt 5:

number the forks 1, 2, 3, 4, 5, each philosopher:

- picks up the fork with the smallest number first, then the larger number second
- if unable to pick up both forks, put a claimed fork down and take a tiny nap

this is a **resource hierarchy** solution – by establishing a **partial order** on resources

- starvation is still possible
- also not practical for large number of resources

dining philosopher problem.. many different solutions, not enough resources and more processes -> allocate resources -> might result in deadlock, livelock (stuff is being done but not making progress), starvation (one or more philosophers never get to eat), race conditions (if we have a bug in the implementation might corrupt resources -> not part of the problem but solution). All these are part of the solution, not the problem (due to design flaw)

Naive algorithm implementation

- implementation of even a naive solution presents problems
- consider algorithm for philosopher 'i':

```
// global variable representing chopstick state
// 0 = unavailable, 1 = available
int chopsticks[5];

while (true)
{
  sleep (s);  // think for s seconds
  while (!chopstick[i] || !chopstick[i+1]); // i+1 modulo 5 arithmetic
  chopstick[i] = false;
  chopstick[i+1] = false;
  sleep (m);   // eat for m seconds
  chopstick[i] = true;
  chopstick[i+1] = true;
}
```

Race condition

Philosopher i = 2:

```
while (true){
  sleep (n);  // thinking
  while (!chopstick[i]
        || !chopstick[i+1]);
  chopstick[i] = false;
  chopstick[i+1] = false;
  sleep (m);   // eating
  chopstick[i] = true;
  chopstick[i+1] = true;
}
```

Philosopher i = 3:

```
while (true){
  sleep (n);  // thinking
  while (!chopstick[i]
        || !chopstick[i+1]);
  chopstick[i] = false;
  chopstick[i+1] = false;
  sleep (m);   // eating
  chopstick[i] = true;
  chopstick[i+1] = true;
}
```

Race condition

```
load R1, [chopstick+3]
cmp  R1 1
```

```
load R1, [chopstick+3]
cmp  R1 1
```

Philosopher i = 2:

```
while (true){
  sleep (n);  // thinking
  while (!chopstick[i]
        || !chopstick[i+1]);
  chopstick[i] = false;
  chopstick[i+1] = false;
  sleep (m);   // eating
  chopstick[i] = true;
  chopstick[i+1] = true;
}
```

Philosopher i = 3:

```
while (true){
  sleep (n);  // thinking
  while (!chopstick[i]
        || !chopstick[i+1]);
  chopstick[i] = false;
  chopstick[i+1] = false;
  sleep (m);   // eating
  chopstick[i] = true;
  chopstick[i+1] = true;
}
```

Consider the following sequence:

```
Thread 1 : load R1, [chopstick+3]  [R1 = 0]
Thread 2 : load R1, [chopstick+3]  [R1 = 0]
Thread 1 : cmp  R1 1
Thread 2 : cmp  R1 1
```

Philosopher i = 2:

```
while (true){
  sleep (n);  // thinking
  while (!chopstick[i]
        || !chopstick[i+1]);
  chopstick[i] = false;
  chopstick[i+1] = false;
  sleep (m);   // eating
  chopstick[i] = true;
  chopstick[i+1] = true;
}
```

Philosopher i = 3:

```
while (true){
  sleep (n);  // thinking
  while (!chopstick[i]
        || !chopstick[i+1]);
  chopstick[i] = false;
  chopstick[i+1] = false;
  sleep (m);   // eating
  chopstick[i] = true;
  chopstick[i+1] = true;
}
```

- depending on the execution order, both philosophers could start eating at the same time
- they both would be inside their critical region

Philosopher with mutex

```
pthread_mutex_t mutex; // a lock, initialized elsewhere

while (true) {
  sleep (s);  // think
  pthread_mutex_lock(&mutex);
  while (!chopstick[i] || !chopstick[i+1]);
  chopstick[i] = false;
  chopstick[i+1] = false;
  pthread_mutex_unlock(&mutex);
  sleep (m);   // eating
  pthread_mutex_lock(&mutex);
  chopstick[i] = true;
  chopstick[i+1] = true;
  pthread_mutex_unlock(&mutex);
}
```

thread 2

thread 1

Deadlock !!!

- **critical section** - part of the program where a shared resource is accessed & may cause trouble
- **mutual exclusion** - ensuring only one process accesses a resource at a time, eg. only one process can enter critical section at any given time
- **mutex/lock** - mechanism to achieve mutual exclusion, two states, queue
- **deadlock** - a state where each process/thread is waiting on another to release a lock → no progress is made
- **livelock** - states of the processes change, but none are progressing
- **starvation** - one process does not get to run at all
- **unfairness** - not all processes get equal opportunity to progress

- **particular API in pthread library**

API	Description
`phread_mutex_init()`	create a mutex
`phread_mutex_destroy()`	destroy a mutex
`phread_mutex_lock()`	lock a mutex, block if already locked
`phread_mutex_trylock()`	lock a mutex, or fail (non-blocking version)
`phread_mutex_unlock()`	unlock a mutex

- pthread_create(thread, attr, start_routine, arg)
  - starts a thread, similar for fork()
- pthread_exit(status)
  - terminates the current thread, similar to exit()
- pthread_join(thread, status)
  - blocks the calling thread until the specified thread terminates, similar to wait()
- pthread_attr_init(attr) and pthread_attr_destroy(attr)
  - initializes / destroys thread attributes
- pthread_atfork()
  - used to register a callback in case fork() is called in a multithreaded program
- pthread_kill(threadid, signal)
  - one thread manually kills a target thread
- pthread_mutex_t mutex
  - initialize a mutex
- pthread_mutex_init(&mutex)
  - create a mutex
- pthread_mutex_destroy(&mutex)
  - destroy a mutex

- pthread_mutex_lock(&mutex)
  - lock a mutex, block if already locked
- pthread_mutex_trylock(&mutex)
  - lock a mutex, or fail (non-blocking version)
- pthread_mutex_unlock(&mutex)
  - unlock a mutex
- pthread_cont_t cond
  - initialize a condition variable
- pthread_cond_init(&cond, &attr)
  - create condition variable
- pthread_cond_destroy(&cond)
  - destroy a condition variable
- pthread_cond_broadcast(&cond)
  - wakes up all threads waiting on condition
- pthread_cond_wait(&cond, &mutex)
  - atomically releases mutex and causes the calling thread to block, until some other thread calls pthread_cond_signal(&cond)
- pthread_cond_signal(&cond)
  - wakes up one thread waiting on cond. If no thread is waiting on cond, the signal is lost
  - must be followed by pthread_mutex_unlock() if the blocked thread uses the same mutex
- pthread_spin_t lock
  - initialize a spinlock
- pthread_spin_init(&lock, int pshared)
  - initialize a spinlock
- pthread_spin_destroy(&lock)
  - destroy a spinlock
- pthread_spin_lock(&lock)
  - lock a spinlock, block if already locked
- pthread_spin_trylock(&lock)
  - lock a spinlock, or fail
- pthread_spin_unlock(&lock)
  - unlock a spinlock
- pthread_rwlock_*()

- pthread share-exclusive locks (also called reader-writer lock)
- Resource supports multiple concurrent readers, but only a single writer

- **Job scheduling algorithm (3 Q)**

First Come First Serve - No Starvation
Round Robin - No Starvation
Shortest Job First - Starvation
Shortest Remaining Time Next - Starvation
Shortest Process Next - Depends on initial prediction
Fair-Share Scheduling - Depends on scheduling algorithm used for each user
Priority Scheduling – Depends

First Come First Serve - No Starvation, Non-preemptive, LEAST context switches
CPU assigned in the order the processes request it, using a FIFO ready queue, new jobs are appended to the ready queue, a running job keeps the CPU until it is either finished, or it blocks.
Disadvantage: CPU-bound process will tie up the CPU, making I/O bound processes run for much longer (convoy effect)

Round Robin - No Starvation, Preemptive
Each process is assigned a time interval, called quantum. If the process exceeds the quantum, the process is preempted, and CPU is given to the next process in the ready queue, while the preempted process goes to back of ready queue.
- very small **Q** implies heavy overhead, but highly responsive system
- very large **Q** implies minimum overhead, but a non-responsive system
- performance of RR depends significantly on the size of the time quantum (**Q**) and the time required for a context switch (**S**)
- **Q** should be large compared to **S**

Shortest Job First - Starvation, Non-preemptive
Sorted on static execution time, based on submitted estimate. Execution time can be dynamically computed based on the history of CPU bursts.
Advantage: Minimum number of context switches
Optimal turnaround time if all jobs arrive simultaneously, minimize average waiting time
Disadvantage: Require advance knowledge of how long a job will execute, has potential for starvation

Shortest Remaining Time Next - Starvation, Preemptive
Next job picked based on remaining time = total time - time already spent on CPU
Ready queue is sorted based on remaining time, remaining time can be static, or dynamically calculated
similar advantages to SJF:

- **optimal turnaround time** even if jobs don't arrive at the same time
- but slightly more context switches

similar disadvantages to SJF:

- requires advance knowledge of how long a job will execute
- has a potential for job starvation
- and also needs to consider cost of context switch (More than N context switches for N processes)

Multi-level queue:

preemptive time-sharing scheduling algorithm that supports process priorities-- ready queue is partitioned into separate queues-- each queue can have a different scheduling algorithm-- a process is permanently assigned to one of the queues scheduling is done based on queues
Multi-level feedback queue:
similar to multilevel queues; but a process **can move** between queues (up or down in priority);
scheduler defined by

- number of queues
- scheduling algorithms for each queue, including different time slices
- when to move a process between queues *

- which queue a process will be assigned when the process needs service

Shortest Process Next - Preemptive
Ready queue is sorted by a predicted next CPU burst, calculated using exponential smoothing:

P = a*B + (1-a)*P'
P = new prediction, initialized to some value (e.g., 0)
P' = previous prediction
a = smoothing factor, commonly ½

Operation Environments	Scheduling Algorithms
Batch systems	• First come, first serve • Shortest job first • Shortest remaining time next
Interactive systems	• Round robin • Shortest process next • Fair share • Lottery • Multilevel queue • Multilevel feedback queue
Real-time systems	• Rate-monotonic scheduling • Earliest-deadline-first

Fair-Share Scheduling -
Fair-share based on owners (users) of the processes. Allocates the CPU among users/groups, instead of processes, all processes belonging to an owner have to share the owner's CPU share.
Priority Scheduling - Non pre-emptive
Lottery Scheduling - Each process gets N lottery tickets based on priority, if ticket picked, process gets time slice. No starvation
Multilevel Queues - ready queue partitioned into separate queues, scheduling depends on queue. Assignment is permanent.
Multilevel Feedback Queues - Like multilevel, but assignment may change . Minimizes starvation, reduces convoy effect

- **deadlock, unsafe state, safe state**

Deadlock ⇒ Unsafe State
Unsafe State !⇒ Deadlock
a system is in a **safe state** if there exists a sequence <P1, P2, …, Pn> of all running processes in the system where they can all finish, while allowing them to claim their maximum resources
a system is in an **unsafe state** if there does not exist such an execution sequence

- when a process requests an available resource, the system determines if granting such request would lead to a safe state --- if new state is safe, request is granted --if new state is not safe, request is denied and process waits

- **cycle + graph + deadlock (for single instance per resource type- use resource-allocation graph) (for multiple instances per resource type- banker's algorithm)**

No cycle ⇒ No deadlock
If cycle:

- one instance per resource type ⇒ guarantee deadlock
- multiple instance per resource type ⇒ possible deadlock

$P=\{P_1,\dots,P_n\}=¿$ set of all PROCESSES in the system. use BOX
$R=\{R_1,\dots,R_m\}=¿$ set of all RESOURCE TYPES in the system. use OVAL
$P_i \rightarrow R_j=¿$ REQUEST EDGE
$R_i \rightarrow P_j=¿$ ASSIGNMENT EDGE

- NO CYCLE ⇒ NO DEADLOCK
- CYCLE:
  - ONE INSTANCE PER RESOURCE TYPE ⇒ DEADLOCK

- OTHERWISE ⇒ CHECK MANUALLY

- **strict alternation (2 threads & running back and forth)**

Two processes, always run the first one for a little bit and then switch to the next one then switch back. One process has to wait for the other one to finish using resources to run.

Drawback: fast process has to wait for the slow one

Strict Alternation - A synchronization approach where two processes alternate entering their critical regions. Uses a global variable `turn = 0`

Drawbacks:

- Busy waiting
- Only works for 2 processes
- No progress - faster process is blocked by slower process

- **Neccesary condition for deadlock**

mutual exclusion - the involved resources are unshareable
hold and wait - a process holding at least one resource is waiting for more resources
no preemption - a resource can be released only by the process holding it (voluntary)
circular wait - there is a cycle

- **Difference deadlock avoicdance, recovery, protection**

- Ensure that the system will never enter a deadlock state (unsafe state) : deadlock **prevention**--- deadlock **avoidance**
- Allow the system to enter a deadlock state and then recover: deadlock **detection** ---- **recovery** from deadlock

Deadlock prevention is what we use to attack the necessary conditions

Deadlock prevention: any technique that attacks one of the 4 necessary conditions.

Avoid mutual exclusion

- mutual exclusion not required for shareable resources (read-only)
- spooling can help for some resource types
- **Peterson's algorithm** - A non-blocking technique for implementing mutual exclusion

Avoid hold and wait condition

- whenever a process requests a resource, it does not hold any other resources

option 1: process must request all needed resources at beginning [lockn(mutex1, mutex2)

- or release resources before acquiring: unlockAndLock

option 2: process can request resources only when it has no resources

Avoid no preemption condition: if a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released, and process is suspended

- preempted resources are added to the list of resources for which the process is waiting
- process will be resumed when it can regain its old & new resources
- only works with resources for which we can save/restore the state (eg. CPU registers)
- complicated mechanism, possible starvation, non-optimal use of resources

```
Thread 1:

unlockAndLock( mutex2, mutex1);
// critical section
unlock( mutex2 );
unlock( mutex1 );
```

Avoid circular wait: most practical condition to avoid --- impose a total ordering on all resources, and require that each process requests resources in an increasing order of enumeration
– lock mutexes in the same order by all threads

Deadlock Example with Lock Ordering

```
void transaction(Account a1, Account a2, double amount) {
   mutex m1, m2;
   m1 = get_mutex(a1); // get exclusive access to account a1
   m2 = get_mutex(a2); // get exclusive access to account a2
   lock(m1);
      lock(m2);
         withdraw(a1, amount);
         deposit(a2, amount);
      unlock(m2);
   unlock(m1);
}
```

Imagine 2 transactions execute concurrently:

first transaction transfers from account A to account B - `transaction("A","B",20)`

second transaction transfers from account B to account A - `transaction("B","A",10)`

⇒ Possible Deadlock.

Deadlock Example with Lock Ordering

```
void transaction(Account a1, Account a2, double amount) {
   mutex m1, m2;
   m1 = get_mutex(a1); // get exclusive access to account a1
   m2 = get_mutex(a2); // get exclusive access to account a2
   if( id(a1) > id(a2))
      swap( m1, m2);
   lock(m1);
      lock(m2);
         withdraw(a1, amount);
         deposit(a2, amount);
      lock(m2);
   lock(m1);
}
```

Alternatively:

```
if( id(a1) > id(a2)) {
    lock(m1); lock(m2);
} else {
    lock(m2); lock(m1);
}
```

Avoiding circular wait condition by ordering resources.

Deadlock Avoidance:

deadlock prevention schemes can lead to low resource utilization

- **deadlock avoidance** can increase resource utilization if some **a priori information** is available
- simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
- a deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

Deadlock detection: Detection algorithm--

** with single instance per resource type-- maintain an up-to-date **wait-for** graph

- nodes are processes
- edge Pi → Pj exists if Pi is waiting for Pj
- can be obtained by **collapsing** resource-allocation graph

- periodically invoke an algorithm that searches for a cycle in the graph
- if there is a cycle, there exists a deadlock

** with multiple instances per resource type:

similar to banker's algorithm

- we try to determine if there is a sequence in which running processes can **finish** executing
- but assumes best case scenario - a process that is given its requested resources will finish without asking for more resources, and then releases all its resources
- does not need Max[] or Need[] matrices

detection algorithms are expensive

- we cannot invoke them on every resource request

- other ideas for invoking detection:
  - check every few minutes
  - check when CPU goes idle

- when, and how often depends on:
    - how often a deadlock is likely to occur?
    - how many processes will be affected?
        - one for each disjoint cycle
- if we check too often - we spend too many CPU cycles on useless work
- if we don't check often enough - there may be many cycles in the resource graph and we would not be able to tell which of the many deadlocked processes “caused” the deadlock

Deadlock recovery: process termination--

<u>process rollback-</u>- programs can be implemented to periodically save their current state (**checkpoint**)

- when restarted, the program detects a checkpoint and resumes computation from last checkpoint (**rollback**)
- programs can checkpoint themselves just before requesting resources

- when deadlock is detected, a program can be terminated and re-scheduled to run later
    - eg. after the other affected deadlocked processes are done
- does not work well with all resource types (eg. printer)
- useful for long computations / simulation

<u>Resource preemption</u>

similar idea to rollback, but instead of checkpointing the program, we checkpoint the resources of the program

- only works with some resource types
- when deadlock occurs:
    - pick a victim process
    - suspend victim process
    - save state of victim's resources
    - give victim's resources to other deadlocked processes
    - when the other processes release the resources, restore resource states
    - return resources to the victim
    - unsuspend the victim

Banker- deadlock avoicdance; Spooling- mutual exclusion

- **Banker algorithm**

show if something is in safe state or not. whether a particular request can be granted or not

- Deadlock detection algorithm

n = number of processes

- m = number of resource types
- Available[] = vector of length m indicates the number of available resources of each type
- Allocation[][] = n x m matrix - current allocation of resources per process

Request[][] = n x m matrix - current requests for resources per process

```
1. Initialize:
   Work = Available
   Finish[i] = (Allocation[i] == 0) for all i = 1,2,3, … n
2. Find an index i such that:
      Finish[i] == false and Request[i] ≤ 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 1 ≤ i ≤ n, then system is not in deadlock
   Otherwise, all processes Pi for which Finish[i] == false are deadlocked
```

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state.

- Memory allocation algorithm (first fit, best fit, next fit, worst fit)

Best Fit - Find the smallest hole that is big enough, leftover (tiny) space becomes new hole
First Fit - Find the first hole that is big enough, leftover space becomes new hole
Next Fit - Same as first fit, but starting at the location of last placement
Worst Fit - Find the largest hole, leftover space is likely to be usable
Quick Fit - Maintain separate lists for common request sizes... Faster search, more complicated management

- Memory fragmentation (internal and external)

Internal Fragmentation - Memory internal to a partition becomes fragmented. (Fixed partitioning-having block size) --- Leads to low memory utilization if partitions are big
External Fragmentation - Memory external to all partitions becomes increasingly fragmented.

Fixed partitioning has both internal and external fragmentation.
Dynamic partitioning only has external fragmentation.
(Dynamic partitioning) leading to low memory utilization.

- Page replacement algorithm (FIFO, LRU, Clock)

FIFO - Replace page that has been in memory for the longest time
can be implemented using a FIFO queue

Optimal - Replaces page that will not be used for the longest period of time
Not practical, since it requires knowing the future
Useful for measuring how well other non-optimal algorithms perform

LRU - Replaces the page that has not been used in the most amount of time

Clock - Approximation of LRU. Uses the reference bit in page table entry, which is automatically set by hardware anytime page is accessed. Frames are organized as a circular buffer. A pointer is maintained pointing to the page to be replace next. If a page has reference bit = 0, replace it. Otherwise set reference bit to 0 and advance pointer to next page.

LEAST PAGE FAULT Optimal < LRU < CLOCK < FIFO MOST PAGE FAULT
==========

Page fault is an exception raised when a process accesses a memory page that is not currently mapped
by MMU, e.g., entry in page table marked invalid

If a program tries to address a page that does not map to physical memory:

1. CPU issues a trap - called *page fault*

2. OS suspends the process
3. OS locates the missing page on disk
   ... If not on disk? Invalid page fault. Results in crash, segmentation fault, core dump, ...
4. OS loads the missing page from disk
5. OS updates the page table
6. OS resumes the process

Global Replacement - OS selects a replacement frame from the set of **all** frames. i.e., one process can steal a frame from another

- Advantage: Greater throughput, so more common
- Disadvantage: Process execution time can vary greatly

Local Replacement - Each process selects only from its own set of allocated frames

- Advantage: More consistent per-process performance
- Disadvantage: Leads to underutilized memory

- Virtual memomory, logical address, physical addresses (same with short answer)

1. **Difference between logical and physical addresses (not the same) doesn't have to be the save sizes but frames and pages have the size. But logical address space is generally bigger than physical address space**

   Logical Address Space - A logical address space is a contiguous space ranging from 0 to MAX, given to the process by the OS.
   As the process executes, the addresses generated by the CPU are logical address spaces.
   If a logical address does not fall into the logical address space range → violation (trap)

   Physical Address Space - A subset of the RAM allocated to a process, containing real memory addresses. Logical addresses are mapped to physical addresses before reaching memory via hardware device called memory management unit (MMU)
   Physical Address Space = set of all mappings from logical addresses

   Logical and physical address spaces does not need to have the same size, but pages (in virtual memory) and frame (in physical memory) must be of the same size
   Logical address space is generally bigger than physical address space

- Copy on Write

  Copy-on-Write (COW) allows parent and child processes to initially share the pages in memory.

  - If either process tries to modify a shared page, the page is copied first, then modified
  - Page table entries need a copy-on-write bit
  - COW allows very efficient process creation (fork) as only modified pages are copied

- Page fault
- Thrashing (process thrashing/ system thrashing)
- file system (generic ones)

2. **File allocation methods and how random access performances are: which ones are better or worse for random access.**
   Contiguous Allocation is best for random access

- file system (inode)

3. **Node question. what do we store in directory entry: file name and pointer to the root iNode.**

   iNode - Basic idea behind indexed allocation is to store a per-file FAT-like structure. Each file has its
   own index blocks, called inodes.

An inode block contains:

- Pointers to blocks belonging to file, or more pointers to even more inodes
- File attributes
  - File size in bytes
  - Device ID
  - Owner
  - Permissions
  - Timestamps
  - Link Count
  - ...

NOTE: iNode does NOT contain a filename

Instead **directory entry** is used to associate filename with the inode

dentry = filename + pointer to inode

Possible to have different file names associated with the same iNode

- differences (contiguous, FAT, inode) Why choose one versus the otherwise
- Disk I/O scheduling
- Semaphore

Semaphores: difference between a counting semaphores and a binary semaphores/mutex, is there a difference between binary semaphores/mutex

Counting Semaphore - A general semaphore representing an integer value S, where S > 0 is the number of processes/threads that can issue a wait and immediately continue to execute

Binary Semaphore - A special case of semaphore where S = 0 or S = 1. When the binary semaphore is locked by a thread, it can be unlocked by **any** thread

Mutex - DIFFERENT FROM BINARY SEMAPHORE, since it can only be unlocked by the locking thread

- **TLB Translation Lookaside Buffer** - A special fast-lookup hardware cache, extremely fast and extremely small (64 to 1K entries). On TLB miss, value is loaded into TLB for faster access next time.

TLB is often implemented as associative memory: hardware capable of fast parallel search based on content. Given a page number, TLB will return the corresponding frame number in constant amount of time.

Short answer: (2 marks/ question)

- Calculate from virtual memory
- Logical address to physical addresses

If a system has a $2^N bytes$ page size, the **page number and the offset** for an address can the found by:

- Convert address to binary
- The LAST N binary digits are the offset
- The rest is the page number

Consider a system with $N\,bit$ logical address space and $x\,byte$ page size. The system supports up to $M\,bytes$ of physical memory.

- Conventional single-level page table:

  $Number\ of\ Entries = 2^N / x$
- Inverted page table:

  $Number\ of\ Entries = M / x$

- Page table (number of page table entries)
- thrashing

when a process is generating too many page faults (very expensive, disks are way slower compared to memory, if page faulting a lot then the program is not executing). spends more time waiting for page faults to be serviced

Thrashing - If a process does not have "enough" pages, the page fault rate is very high. A thrashing
process is a process that is progressing slowly due to frequence page swaps.
Many thrashing processes → low CPU utilization
→ OS may think it needs to increase the degrees of multiprogramming ... so OS adds another process to the system making things even worse.

Techniques to reduce thrashing -> know those

Local Page Replacement - When a process is thrashing, OS prevents it from stealing frames from other processes. At least the thrashing process cannot cause the entire system to thrash

Working Set Model - OS keeps track of pages that are actively used by a process (working set), the working set of a process changes over time, and the OS periodically updates the working set for each process, using a moving time window

Page Fault Frequency - Establish acceptable bounds on page fault rate.

- If actual page fault rate of a process too high → process gains a frame
- If actual page fault rate of a process to low → process loses a frame

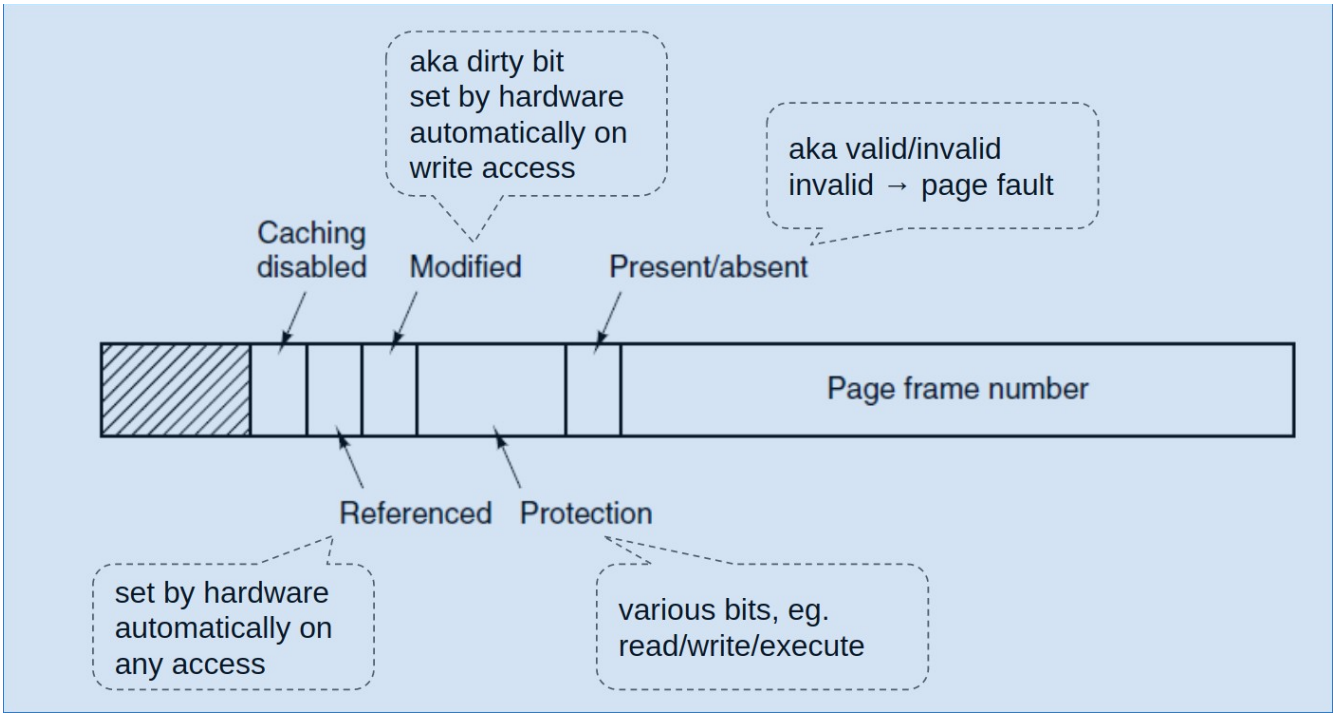
- Memory allocation

- Page table entries (what do they contain? With the bit)

know all the different bits an entry contains: permission +invalid/valid+dirty bit+pointer to the frame …

A page entry contains ...

- **Page Frame Number -** corresponding frame number
- **Present/Absent Bit -** valid/invalid bit.
  - Invalid ⇒ Page fault
  - Valid ⇒ frame in physical memory
- **Protection Bits -** various bits... e.g., read/write/execute access
- **Modified (Dirty) Bit -** dirty bit set by hardware automatically on write access
- **Referenced Bit -** set by hardware automatically on any access
- **Caching Disabled Bit -** caching disabled?



- **Generic schedule (preemptive, non-preemptive)**

non-preemptive — context switch happens only voluntarily

- multitasking is possible, but only through cooperation
- process runs until it does a blocking syscall (eg. I/O), terminates, or voluntarily yields CPU
- example: FCFS

preemptive — context switch can happen without cooperation

- usually as a direct or indirect result of an interrupt, but not limited to clock interrupt

  eg. new job is added, existing process is unblocked
- example: SRTN

preemptive time-sharing — special case of preemptive

- processes are context switched periodically, usually to enforce time-slice policy
- implemented through clock interrupts
- without a clock, only cooperative multitasking (non-preemptive) is possible
- example: RR

response time: how long before you get first feedback, often response = start – arrival

turnaround time: time from arrival to finish, turnaround = finish – arrival

CPU time: how much time the process spent on CPU

waiting time: total time spent it waiting queue, waiting = turnaround - CPU - I/O

- Generic I/O

1. **character devices (mouse, keyboard, printer-> access sequentially) and block devices (disks) and other (timer)**

   Block Devices - Store information in fixed-size blocks

- Each block has its own address
- Data transferred in units of one or more entire blocks
- Read or write can be done in any order
- *E.g., HARD DISK, CD-ROM, USB*

Character Devices - Delivers or accepts a stream of characters, without regard to any block structure

- NOT addressable, and no seek operations
- *E.g., PRINTER, NETWORK INTERFACE, MOUSE, KEYBOARD*

Other Devices - Clock ... Also known as time

- Fork

2. **creates a process by duplicating the current process. Returns a value, negative (failed, doesn't kill the parent), 0(child) or positive (parent). The parent would know the process id of the child because that's the return value of fork , child will have to figure it out in some other way**

pid_t fork(void)

fork() creates a new process by duplicating the calling process. The child process and the parent process run in separate memory spaces. At the time of fork() both memory spaces have the same content.

Memory writes, file mappings, and unmappings performed by one of the processes do not affect the other.

The child process is an exact duplicate of the parent process except:

- The child has its own unique PID
- The child's parent PID is the same as the parent's PID

Example:

```
int main()
{
    /* create & run child process - a duplicate of parent
     * but remember the return value */
    pid_t  pid = fork();
    /* both parent and child will execute the next line,
     * but will have different value for pid:
     *      0 for child
     *            non-zero (the child's pid) for parent */
    printf("My pid is %d.\n", pid);
}
```

- Premption and context switch

Reasons for preemption: time slice, I/O, page fault.

- Time slice exceeded
- New job added
- Existing process is unblocked
- I/O
- Page fault

- Low level atomic (swap, test & set, CAS)

Compare and swap is a technique used when designing concurrent algorithms. Basically, compare and swap compares an expected value to the concrete value of a variable, and if the concrete value of the variable is equals to

the expected value, swaps the value of the variable for a new variable. Compare and swap may sound a bit complicated but it is actually reasonably simple once you understand it, so let me elaborate a bit further on the topic.

```
boolean Test_and_Set( boolean memory[m] )
{ [
    if( memory[m] )          // lock denied
        return True;
    else {                   // lock granted
        memory[m] = True;
        return False;
    }
  ]
}
```

Test_and_Set is a special assembly language instruction that does two operations autonomously. That is, the instruction can not be interrupted in the middle and it is not necessary to disable interrupts. *Test_and_Set* (TS) is a privileged instruction requiring supervisory mode permissions.

Names of the atomic operations used for synchronization

Atomic Operation - An operation that appears to execute instantaneously to the rest of the system.

Cannot be interrupted by signals, threads, interrupts, ...

Examples: assignment, `s++`, `s--`

- `pthread_mutex_lock(&mutex)`
  - lock a mutex, block if already locked
- `pthread_mutex_trylock(&mutex)`
  - lock a mutex, or fail (non-blocking version)
- `pthread_mutex_unlock(&mutex)`
  - unlock a mutex
- `sem_wait(&sem)`
  - suspends calling thread until semaphore is non-zero, then atomically decreases the semaphore count
- `sem_post(&sem)`
  - atomically increases the semaphore, never blocks, may unblock blocked threads, safe to use in signal handlers in Linux on 486+ hardware
- `pthread_cond_wait(&cond, &mutex)`
  - atomically release mutex and cause the calling thread to block, until some other thread calls `pthread_cond_signal(&cond)`

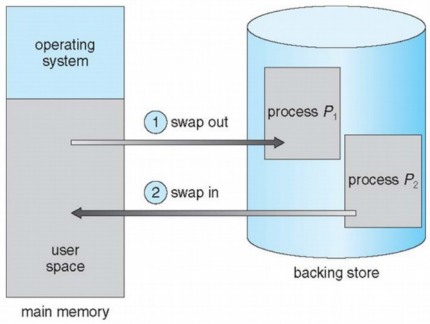
- swapping question (different type of swap)

Swapping memory (for process)

Difference between paging and swapping: paging(more efficient) replaces a single page at a given time, swapping replaces all the memory at the same time (swaps the entire process out)

Swapping replaces all memory at the same time. i.e., swaps the entire process out
Paging replaces a single page at a given time (MORE EFFICIENT)

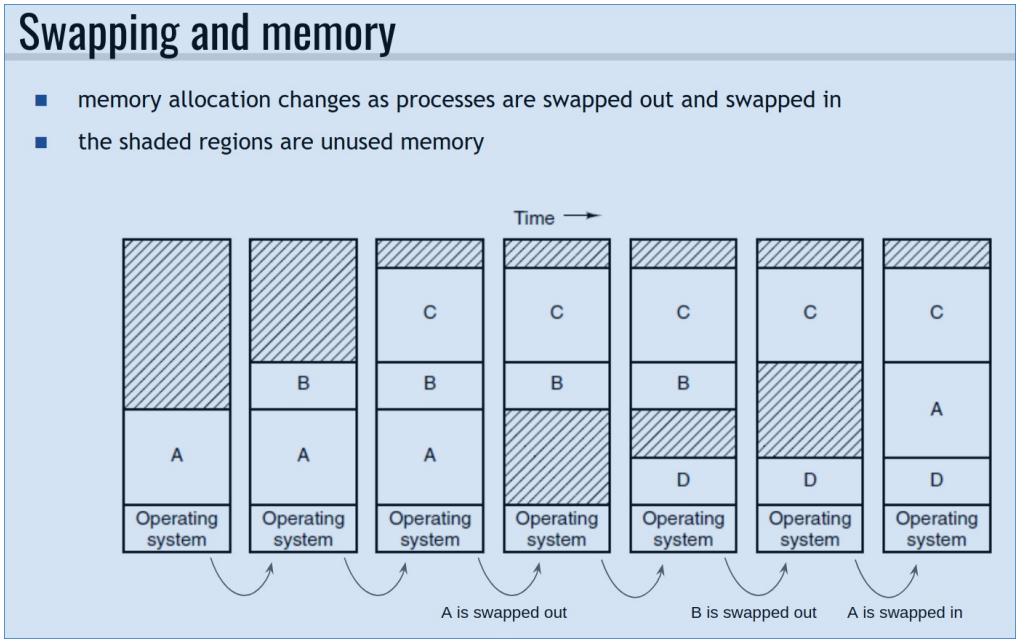
- a process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **backing store** - fast storage large enough to accommodate copies of all memory images for all processes
- swapping allows the OS to run more processes than the available physical memory



Does the swapped out process need to swap back into the same physical addresses?

- depends on address binding method
- much easier if MMU is used

    - must be careful with pending I/O, especially when using memory-mapped device registers
    - I/O results could be sent to kernel, then to the process (double-buffering)
- warning: context switch time can be extremely high
- standard swapping not used in modern operating systems
- although modified versions of swapping are used on many systems (eg. Linux, Windows)
    - swapping is disabled initially
    - swapping enabled if more than threshold amount of memory allocated
    - swapping is disabled again once memory demand reduced below threshold

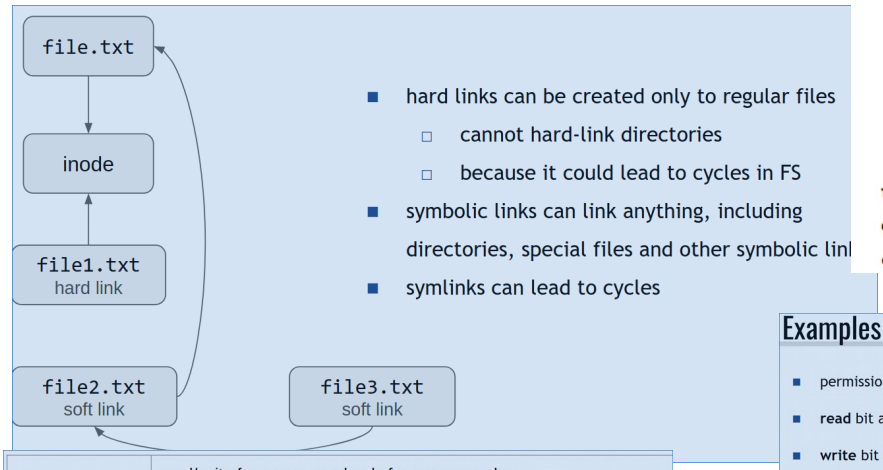


- bash

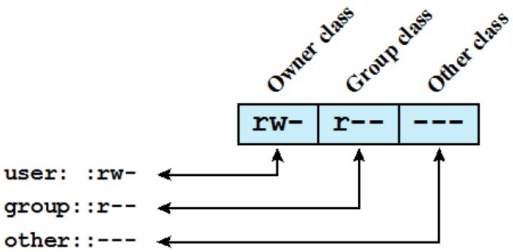
```
find . -type -f -name "*$1" -printf '%p %s\n' |
sort -k 2 -t' ' -n -r |
head -n $2 |
awk '{x += 2; print $1, $2}END{print"Total size:"x}''
```

alphabetic sorting file on the basis of the 2nd field:	`sort -t"," -k 2 file`
numeric sorting file on the basis of the 2nd field:	`sort -t"," -k 2 -n file`
numeric sorting file on the basis of the 2nd field, reversed order:	`sort -t"," -k 2 -n -r file`
Print all lines in file.txt that are longer than 80 characters:	`cat file.txt | awk 'length($0) > 80'`
Print the sizes of each file in the current directory, and the total size:	`ls -l files | awk '{ x += $5 ; print $5 } END { print "total bytes: " x }'`
To print the first 3 lines of the file readme.txt:	`$ head -3 readme.txt` Or you can use pipes: $ cat readme.txt | head -3
Find all files in the current directory, but only print the first 10:	`$ find . -type f | head -10`

o find and print all files in the current directory:	$ find . -type f -print
To find all files that start with a prefix 'abc', and print their full paths	$ find . -type f -name 'abc*' -print
Find all files in the current directory and print them together with their sizes	$ find . -type f -printf '%p %s\n'
scan.sh jpg 5 ./face.jpg 88374	find -type f -name "*.$1" -printf "%p %s\n" | sort -k 2 -n -r | head -$2 | awk '{print $1 " " $2; total += $2} END { print "Total size: " total }'
create hard-link file1.txt to file.txt	**ln file.txt file1.txt** hard link points to the same inode if we delete file.txt, file1.txt will still work
soft link points to a filename if we delete file.txt, file2.txt will be broken	**ln -s file.txt file2.txt**



- hard links can be created only to regular files
  - cannot hard-link directories
  - because it could lead to cycles in FS
- symbolic links can link anything, including directories, special files and other symbolic lin
- symlinks can lead to cycles



-rw-r--r--	read/write for owner, read-only for everyone else
-rw-r-----	read/write for owner, read-only for group, forbidden for everyone else
-rwx--x--x	read/write/execute for owner, execute-only for everyone else
-r--r--r--	ready-only for everyone
-rwxrwxrwx	read/write/execute for everyone (bad idea)
----rwxrwx	yes it's possible, owner has no rights, everyone else does...

Examples - permissions for directories

- permission bits are interpreted slightly differently for directories
- **read** bit allows listing of file/directory names
- **write** bit allows creating aand deleting files in directory
- **execute** bit allows entering the directory and getting attributes of files in the directory
- not all combinations make sense: eg. read without execute

drwxr-xr-x	all can enter and list the directory, only owner can add/delete files
drwxrwx---	full access to owner and group, off limits to world
drwx--x--x	full access to owner, while group & others can access only known files
drwxrwxrwx	anyone can do anything

`chown -R root dir1`	changes owner of dir1 to root, and recursively everything inside dir1
`chmod g+w,o-rwx f1 f2`	adds group write access to files f1 and f2, and removes all access to f1 and f2 for the world
`chmod -R o-rwx .`	removes access for the world to current directory and everything inside it (recursively)
`chmod u+rw,g+rw,u-x,g-x,o-rwx f1`	f1 will allow read/write to owner & group, everyone else will have no access
`chmod 660 f1`	same as above but makes you look “pro”
`chmod +x f1`	f1 will be executable to everyone
`chmod og-rwx f1`	disable all group/world access from f1

- FAT (file allocation)

File Allocation Table - is a variation of linked allocation. The 'next' pointers are stored separately in a table, beginning of file system volume has a table indexed by block number. Table contains all pointers, one for each block.
There is one FAT for the entire disk.
Directory entry contains index into FAT
Much like a linked list, but all pointers are stored together ... faster and cache-able, new block allocation is simpler

A consistent FAT cannot:

- have too many or too few blocks for a file
- Blocks allocated to file cannot contain a cycle
- Files cannot share blocks with any other file

Advantages: Easier random access
Disadvantages: The entire table must be in memory at all times to achieve efficient random access
Table can be quite big for large disks

- Paging question, paging algorithm
- Effective access time handling EAT

Symbolic links and hard links: hard link: two or more files point to the same inode. Symbolic links point to the file name

Symbolic Links - Can link anything, including directories, special files, and other symbolic links
link points to the file name

Hard Links - Can be created only to regular files
cannot hard-link directories because it could lead to cycles in filesystem
Can have different filenames associated with the same inode (called hard links)

Effective memory access time calculation - Know formula

$p = ¿$ probability of page fault or page fault rate
$ma = ¿$ memory access time
$pfst$ = page fault service time

$$EAT = Effective\ Access\ Time = (1 - p) * ma + p * (pfst + ma)$$

WITH TLB: $EAT = (1 - p) * (tlbs + 2 * ma) + p * (tlbs + ma)$
p = probability of TLB hit
$tlbs$ = TLB search time

Long answer (8 marks/ question)

- File allocation
- Page replacement (Reference string, the algorithm to use LRU, Opt, Clock), how many page fault
- Page table question (calcualte about virtual memory)
- Convert from physical addresses to logical addresses and vice versa
- I/O scheduling question (the algorithm)
- Inode question (indirect pointer) – calcualte to maximize the file size supported

iNodes: indirection pointers (double, triple)! and do calculations

Example: Block size 1kb, block address 4 bytes
Single inode with 12 direct entries

→ max file size 12 KB

Adding single indirect

1KB block can have 1KB/4B = 256 entries

→ max file size = 256 + 12 blocks = 268 KB

Adding double indirect

or 256 blocks each with 256 addresses

→ max file size = 2^8*2^8 + 12 ~ 64 MB

Adding triple indirect

→ max file size = 2^(3*8) + 12 ~ 16 GB

1. **How big of files are supported by ext.???**

ext2 Max File Size = 2TB
ext3 Max File Size = 2TB
ext4 Max File Size = 16TB

- Deadlock (single instance per resource type)
- Calculate safe state (banker algorithm)
- GANTT chart (for the schedule)
- Multi-thread code (what is possible outcome, race condition, what is the possible out come)

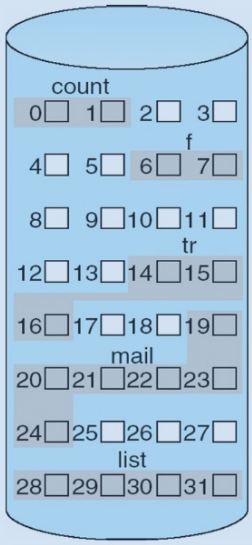
---

1. File Allocation Method
-

contiguous allocation – each file occupies set of contiguous blocks

- best performance in most cases
- simple – only starting location (block #) and length (number of blocks) are required
- problems include
  - finding space for file,
  - knowing file size at creation,
  - external fragmentation after file deletion,
  - need for compaction off-line (downtime) or on-line (reduced performance)
    - aka defragmentation
- not very common
- useful for tapes & read-only devices such as CD-ROMs

Contiguous allocation



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

- mapping from logical to physical address: assuming block size is a power of 2

logical address:

q	r

q = upper bits

r = lower bits

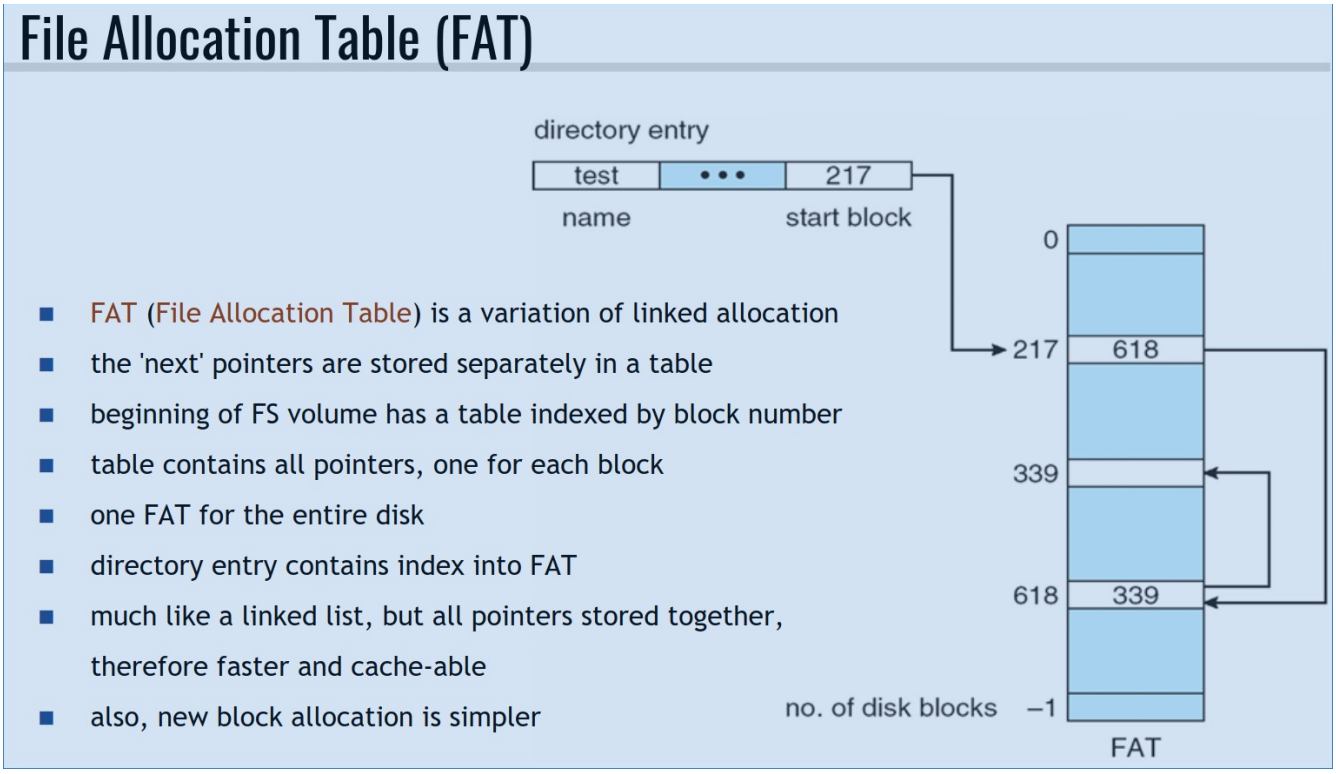
physical address computation:

block = q + "address of first block"

displacement within block = r

linked allocation each file is stored in a linked list of blocks

- each block contains file content, plus a pointer to the next block
- file ends at a block with NULL pointer
- no external fragmentation → no compaction needed
- separate free space management needed - eg. linked list of free blocks
- reliability can be a problem – imagine losing a block due to disk failure
- major problem: locating a block can take many I/Os and disk seeks
    - logical address to physical address mapping requires traversing the list
    - we could cache the 'next' pointers, but would still need to read entire file first
- we could improve efficiency by clustering blocks into larger groups but that increases internal fragmentation



Advantages:

- easier random access

- Disadvantages:
  - the entire table must be in memory at all times to achieve efficient random access
  - table can be quite big for large disks

INODES dea behind indexed allocation is to store a per-file FAT-like structure

- each file has its own index block(s), called inodes
- an inode block contains:
  - pointers to blocks belonging to file, or more pointers to even more inodes
  - plus various file attributes:
    - file size in bytes, device ID, owner, permissions, timestamps, link count, …
- notice - inode does not contain a filename
- instead directory entry is used to associate filename with the inode
  - dentry = filename + pointer to inode
  - possible to have different filenames associated with the same inode: call hard links

inodes in Linux (ext2)

- example: block size 1KB, block address 4 bytes
- single inode with 12 direct entries

  → max file size 12KB
- adding single indirect pointer to inode:

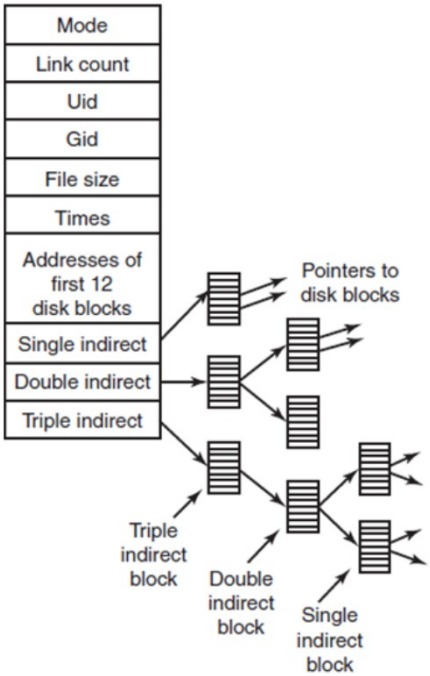
  1KB block can have 1KB/4B=256 entries

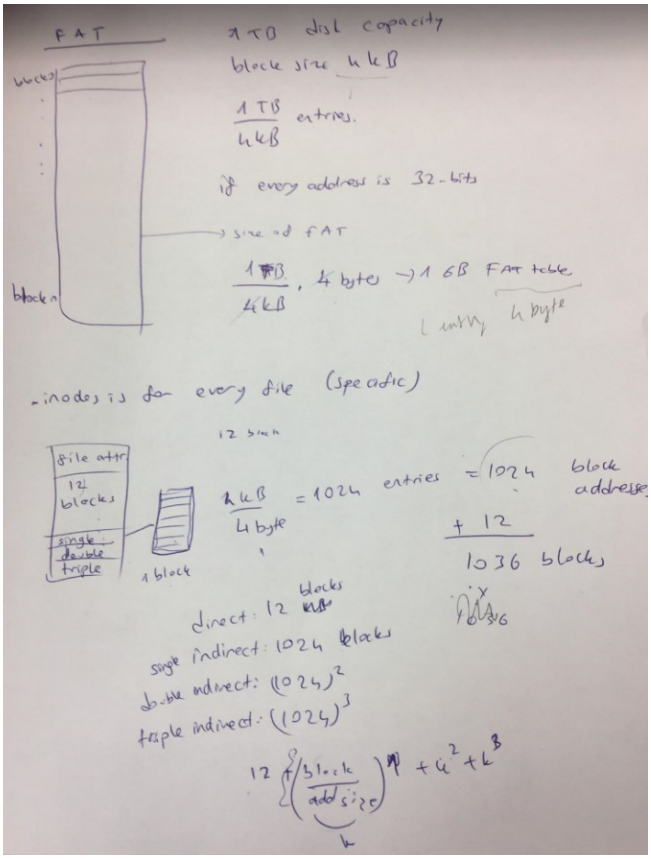
  → max file size 256+12 blocks = 268KB
- adding double indirect pointer as well:

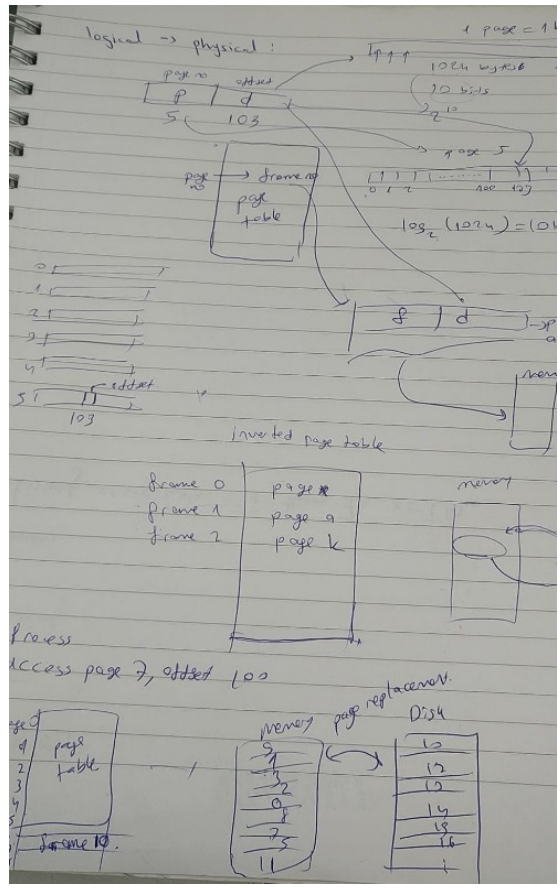
  or 256 blocks each with 256 addresses

  → max file size ~ 2^{16} blocks ~= 64MB
- adding triple indirect pointer:

  → max file size ~ 2^{24} blocks ~= 16GB
- ext3 max file size = 2TB
- ext4 max file size = 16TB (using 48bit addresses)

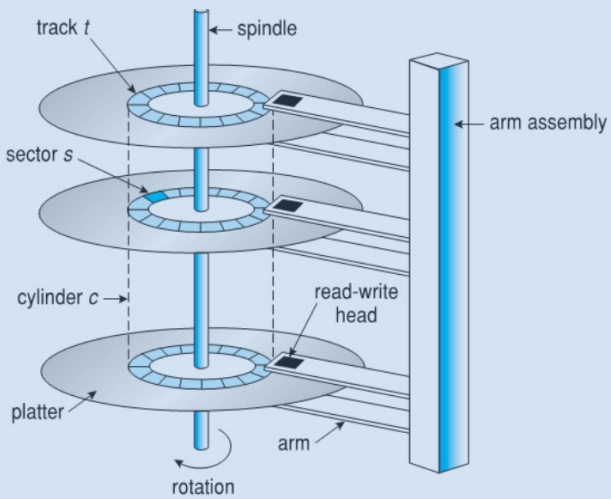






DISK I/O physical description:

Magnetic disks



Physical description:

- each disk platter has a flat circular shape (1.8” < diameter < 5.25”)
- platters rotate
- the read-write heads *fly* just above the surface of each platter
- head crash: the head makes contact with the disk surface, causing permanent damage to the disk
- each head is attached to a disk arm that moves all heads at the same time

- low-level format or physical format: writes low level information to the disk, dividing it into series of tracks, each containing some number of sectors, with small gaps between the sectors
- components of a sector:
  - preamble: starts with a special bit sequence, cylinder number, sector number, etc.
  - data: depends on the format (eg. 512 bytes)
  - ECC: error correction code - contains redundant information to be used for detecting and correcting read errors
- the formatted capacity is about 20% lower than the unformatted capacity

Preamble	Data	ECC

a disk sector

OS can improve the overall I/O performance by reordering disk I/O requests

- disk I/O scheduling tries to minimize total head movement
- FCFS scheduling
- SSTF scheduling
- elevator scheduling
  - SCAN, C-SCAN, LOOK, C-LOOK

FCFS	Fait	but it generally does not provide the fastest overall service
Shortest Seek Time First SSTF (sort the input; check the gap)	• seek time = distance to move the heads	may cause starvation of some requests
SCAN (elevator) scheduling the head continuously scans back and forth across the disk and serves the requests as it reaches each cylinder head moves all the way to first/last cylinder before turning back [Sort then align then figure which direction to go]		requests at either end tend to wait the longest
Look scheduling nearly identical to SCAN, but head does not move all the way to first/last cylinder before turning back	• instead it only goes as far as necessary • results in the same request order as SCAN, but less overall head movement	
C-SCAN same as SCAN in one direction • but after reaching last cylinder, head repositions to the first cylinder • more uniform wait time than SCAN		
C-LOOK small optimization of C-SCAN, head only goes as far as needed by the next request (same optimization as SCAN → LOOK)		Less consistent wait time

Disk scheduling

- the performance of a scheduling algorithm depends on:
  - the number and types of requests
  - the file-allocation method
  - the location of directories and index blocks
- either SSTF or LOOK is a reasonable choice for a default algorithm
  - C-LOOK if we need more consistent wait times
- other scheduling algorithms also consider:
  - rotational latency
  - priority of the task - requests belonging to higher priority process receive higher priority
    eg. requests related to demand paging should receive higher priority
  - prioritize read over write, since read requests usually block processes
  - examples: completely fair queuing (CFQ) & deadline scheduler on Linux

BANKER ALGORITHM: Banker's algorithm tries to find whether there is an execution sequence that would finish all running processes, assuming worst case scenario - that each process requests its maximum declared allocation as the next step.

- another avoidance algorithm
- more general than resource-allocation graph algorithm
- works with multiple instances per resource type
- requirements:
  - each process must declare maximum resources at the beginning
  - when a process requests a resource it may have to wait (even if resource is available)
  - when a process gets all its resources it must return them in a finite amount of time

Let `n` = number of processes, and `m` = number of resources types.

- **Available** - vector of length *m*
  - `Available[j]=k` means there are *k* instances of resource type R_j available
- **Max** - matrix *n x m*
  - `Max[i,j]=k` means that process P_i may request at most *k* instances of resource type R_j
  - `Max[i]` represents the i-th row of Max
- **Allocation** - matrix *n x m*
  - `Allocation[i,j]=k` means that process P_i is currently allocated *k* instances of R_j
- **Need** - matrix *n x m*
  - `Need[i,j]=k` means that P_i may need *k* more instances of R_j to complete its task
- Note: `Need[i,j] = Max[i,j] – Allocation[i,j]`
  - only 2 of Need/Max/Allocation are needed, the 3rd can be computed on the fly

```
1.  Initialize temporary vectors:
    Work = Available ; vector copy
    Finish[i] = false for i = 0, 1, …, n-1
2.  Find an i such that
      Finish[i] = false AND Need[i] ≤ Work ; vector comparison
    If no such i exists, go to step 4
3.  Update Work and Finish:
      Work = Work + Allocation[i] ; vector sum
      Finish[i] = true
    go to step 2
4.  If Finish[i] == true for all i, return true ; system is in a safe state
    else return false ; system is not in a safe state
```

Complexity:

O(n*n*m)

Algorithm to determine if request can be granted for process P_i

Inputs: (Request, i)

- Request is a vector - where Request[j] = k means process P_i wants k instances of R_j
- process number i

Algorithm:

- If Request[] > Need[i] ⇒ raise error, process has exceeded its maximum claim
- If Request[] > Available ⇒ then P_i must wait, since resources are not available
- Pretend to allocate Request[], and see if the state is still safe:
  - Save current state of Available, Allocation and Need
  - Available = Available - Request
  - Allocation[i] = Allocation[i] + Request
  - Need[i] = Need[i] - Request
- check if state still safe using Banker's algorithm
- If state safe ⇒ the request can be granted (resources are allocated to P_i), discard the saved state
- If unsafe ⇒ request must block, P_i must wait, restore the saved state

RAID – redundant array of independent disks- multiple disk drives provide reliability via redundancy

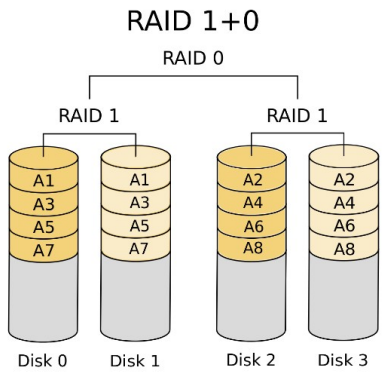
Raid 0: no backup
Raid 1: keeps 1 or more duplicates of a disk; with N disks, it is tolerant to N-1 simultaneous disk failures
Raid 4: one disk dedicated to contain parity information; with N disks, only N-1 are used for data
write is slow, since parity disk is a bottleneck

Raid 5: similar to RAID 4, but parity information is distributed among all disks; with N disks, only N-1 space is used for data

Raid 6: similar to RAID 5, but doubles the amount of parity (parity computation more complicated) with N disks, only N-2 space is used for data

RAID 1+0 – striped mirrors

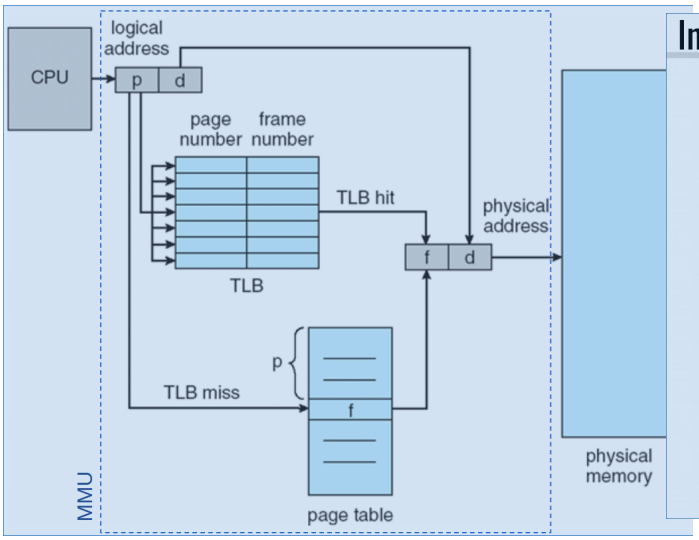
- aka RAID 10, is an example of a hybrid/nested RAID
  - nests RAID 1 in RAID 0 configuration
- purpose: very fast & very reliable
  - combines advantages of RAID 0 and RAID 1
  - only pure RAID 0 performs better
- in simplest form (4 disks), it can survive at least 1 disk failure, and if lucky 2 failures
- common for high-performance uses where data cannot be lost, eg. databases, email server



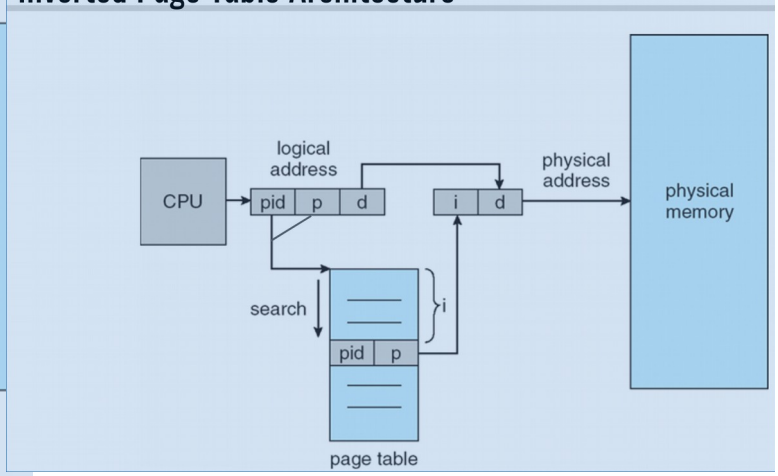
I/O Devices

- Block devices:
  - store information in fixed-size blocks (eg.512 bytes to 32KB)
  - each block has its own address
  - data transferred in units of one or more entire blocks
  - read or write can be done in any order
  - e.g., hard disk, CD-ROMs, USB
- Character devices:
  - delivers or accepts a stream of characters, without regard to any block structure
  - not addressable, and no seek operations
  - e.g., printer, network interface, mouse, keyboard
- Other devices:
  - clocks (also known as timers)

PAGING

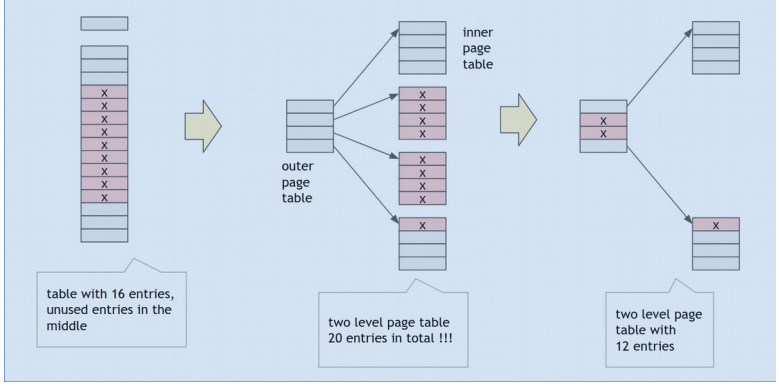


Inverted Page Table Architecture



Two-Level Page-Table Example

- a simple technique is a two-level page table – essentially we page the page table
- no need to store the small pages with all entries invalid …



Two-Level Page-Table Example

- a 32-bit logical address with 4K page size is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits

page #	*offset*

- since the page table is paged, the page number is further divided into:
  - a 10-bit outer page number *p1*
  - a 10-bit inner page number *p2*
- the logical address is then:

page number		page offset
p_1	p_2	d
10 bits	10 bits	12 bits

  - *p1* is an index into the **outer page table**
  - *p2* is an index into an **inner page table**
  - also known as **forward-mapped page table**

Multi-level page table example

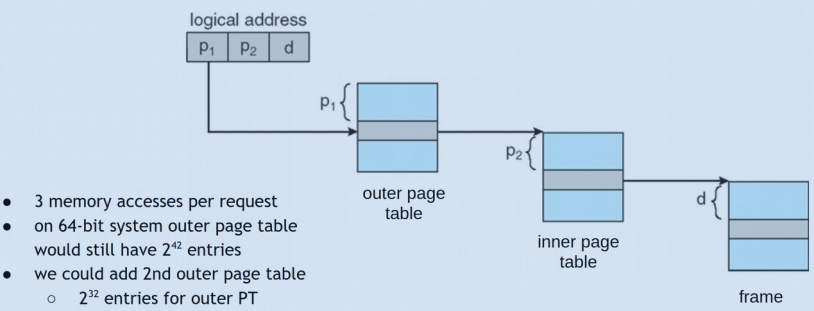
- consider a 64-bit system, with 4KB page size (2^{12} bits), and 8 bytes per entry
- single page table → 2^52 entries x 8B/entry → 2^55 bytes → ~36 petabytes
- how many page table levels do we need if we want each page table fit inside a page?
  - a page can fit 4KB/8B = 512 entries = 2^9 entries
  - with 12-bit offsets, that means we need [52bits/9bits-per-level] = 6 levels!!!

p_1	p_2	p_3	p_4	p_5	p_6	d
7 bits	9 bits	9 bits	9 bits	9 bits	9 bits	12 bits

  - with 6-level hierarchical PT each memory request would require 7 memory accesses
    - 6 accesses for translating logical address → physical address
    - 1 access for the actual memory location
  - this would be way too slow

Chap 15: deadlock

Address-Translation Scheme



- 3 memory accesses per request
- on 64-bit system outer page table would still have 2^{42} entries
- we could add 2nd outer page table
  - 2^{32} entries for outer PT
  - 4 memory accesses per req.

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Example of Banker's Algorithm

- 5 processes:
  - P0, P1, P2, P3 and P4
- 3 resource types:
  - A (10 instances), B (5 instances), and C (7 instances)
- Current state snapshot:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2	7 4 3
P1	2 0 0	3 2 2		1 2 2
P2	3 0 2	9 0 2		6 0 0
P3	2 1 1	2 2 2		0 1 1
P4	0 0 2	4 3 3		4 3 1

Need = Max - Allocation

Is the system in safe state?

Yes - safety algorithm
P1, P3, P4, P2, P0

Banker's algorithm example

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	2 3 0	7 4 3
P1	3 0 2	3 2 2		0 2 0
P2	3 0 2	9 0 2		6 0 0
P3	2 1 1	2 2 2		0 1 1
P4	0 0 2	4 3 3		4 3 1

Given the state of the system on the left, can we grant an additional request (1,0,2) to P1?

1. Update Allocation and Available
2. Calculate Need
3. Is the new state safe?

Use Banker's algorithm, which produces sequence:

1. P1: available = (2,3,0) + (3,0,2) → (5,3,2)
2. P3: available = (5,3,2) + (2,1,1) → (7,4,3)
3. P0: available = (7,4,3) + (0,1,0) → (7,5,3)
4. P2: available = (7,5,3) + (3,0,2) → (10,5,5)
5. P4: available = (10,5,5) + (0,0,2) → (10,5,7)

Answer: **yes**, we can grant the request. Possible execution sequence after granting the request:
P1, P3, P0, P2, P4

Example of Banker's Algorithm

- 5 processes:
  - P0, P1, P2, P3 and P4
- 3 resource types:
  - A (10 instances), B (5 instances), and C (7 instances)
- Current state snapshot:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2	7 4 3
P1	2 0 0	3 2 2		1 2 2
P2	3 0 2	9 0 2		6 0 0
P3	2 1 1	2 2 2		0 1 1
P4	0 0 2	4 3 3		4 3 1

Final exam question candidates:

- Can request (1,0,2) for P1 be granted?
- Can request (3,3,0) for P4 be granted?
- Can request (0,2,0) for P0 be granted?

Show your work / show possible sequence.

Example of Detection Algorithm

- Five processes P0 through P4, and three resource types A, B, C:

  A with 7 instances, B with 2 instances and C with 6 instances
- Snapshot at time T:

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

- Q: Is there a deadlock?
- A: No, possible sequence <P0, P2, P3, P1, P4> will result in Finish[i] = true for all i

- P2 requests an additional instance of type C, i.e. (0,0,1)

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 1	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

  - State of system?
    - P0 can run and finish
    - system reclaims resources held by process P0
    - but insufficient resources to fulfill any other request
    - deadlock exists, consisting of processes P1, P2, P3, and P4

Address translation scheme

- Address generated by CPU is divided into:
  - **Page number** (p) - used as an index into a page table which contains base address of corresponding frame in physical memory
  - **Page offset** (d) - combined with base address to define the physical memory address that is sent to the memory unit

page number	page offset
p	d
m -n	n

m-bit logical address space (2^m), and n-bit page size (2^n)

Example: 16-bit logical address **1010101001100101b**, and 2^{10} page size

6-bit page number	10 bit page offset
101010b	1001100101b

Demand paging performance

- paging performance is commonly evaluated by the **effective access time** for memory access
- Let:
  - **p** = probability of page fault or **page fault rate** (0 <= p <= 1)
    - **p = 0** → all pages are in memory, no page fault
    - **p = 1** → all pages are on disk, all memory accesses are page faults
  - **ma** = memory access time
  - **pfst** = page fault service time, ie. how long does it take to service a page fault
- Then:
  - effective access time: EAT = (1-p) * ma + p * (pfst + ma)
- Non-realistic example: calculate EAT if page fault probability is 50%, ma = 1m

~101 x slower

  - EAT = (1-0.5) * 1ms + 0.5 * (10ms + 1ms) = 0.5ms + 5.5ms = 6ms
- Realistic example: calculate EAT if page fault probability is 1/1000, ma = **100ns** and pfst = **10ms**
  - EAT = (1-0.001) * 100ns + 0.001 * 10,000,100ns ~= **10099.9ns**

- Why is the length of the time quantum important for RR? too short waste nothing (CPU circle do too long, no interact system)
- Will processes experience starvation when using RR scheduling? no. not possible
  Yes or No
- In a multi-level feedback queue, processes are guaranteed to be free from starvation. depended. (most of is true) but it does not have to be
  True or False or It Depends
- When using user-level threads, a blocking system call will suspend the entire process?
  True or False

- virtual address space = 64KB
- physical address space = 32KB
- page size = 4KB
- calculate:
  - frame size = **4KB** (same as page size)
  - # of pages = **16** (64KB / 4KB)
  - # of frames = **8** (32KB / 4KB)

Non-preemptive (SJF, FCFS); Preemptive (SRTN)

A. First come, first serve (FCFS)
B. Shortest job first (SJF)
C. Shortest remaining time next (SRTN)

Assuming no I/O:

- Which one of the above scheduling algorithms achieves the optimal average turnaround time? C
- Which one of the above scheduling algorithms may involve more than N context switches for N processes? C
- Identify the nonpreemptive algorithms and the preemptive algorithms. B

Review

CPSC 457

- Define deadlock. state, processes
- If there is a deadlock, that means there is a circular wait between processes. True or False
- If there is a circular wait between processes, than means there is a deadlock. True or False general
- Which of the following methods is used to prevent circular waiting among processes and resources?
  - Spooling
  - Request all resources at the beginning
  - Take resources away
  - Order resources & lock in order
- How do we detect a deadlock? banker's graph
- Name three approaches for deadlock recovery.
- What is a checkpoint? save state of an application.

- Name two registers used in a simple MMU implementation.
- Explain logical address / physical address.

- Which file block allocation scheme suffers from external fragmentation? internal fragmentation – both
  - Contiguous or Linked
- Describe the main difference between FAT and inode. system ...
- After deleting a file, all hard links to the file will report an error when accessed.
  True or False
- After deleting a file, all soft links to the file will report an error when accessed.
  True or False

- Which one of the following disk scheduling algorithms could lead to starvation among requests?
  - FCFS
  - SSTF
  - SCAN
- Which RAID configuration cannot survive a single disk failure?
  - RAID 0
  - RAID 1
  - RAID 5
  - RAID 10

  at least 1 disk failure can afford

  2 disk fails (1, 10)
- Keyboard is a character device.
  - True or False

1KB= 2^{10} Byte; 1MB= 2^{20} Bytes; 1 GB= 2^{30} Bytes; 1TB= 2^{40} Bytes