457 Final exam: slides, assignments (cumulative, study everything)

3 sections: MC, short answer, long answer (sometimes simple calculation)

MC (30):

1. dining philosopher problem.. many different solutions, not enough resources and more processes -> allocate resources -> might result it 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)
2. **Pthread API -> what does the functions do ? e.g., pthread\_mutex\_lock, locks a mutex that is not yet locked and continues to execute. Thread will block and doesn’t continue to execute if the lock is already locked**

* 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

1. **RR scheduling algorithm and time quantum/slice. Pros and cons with choosing big/small quantum. What if time slice was one year: each process will get a year to run -> no context switch will happen until each process finished executing -> this RR will become FCFS**

**RR**: A preemptive (involuntary context switch) version of FCFS. Each process is assigned a time

interval, called quantum. If the process exceeds the quantum, the process is preempted (context switch), and CPU is given to the next process in ready queue, while the preempted process goes to back of ready queue.

**Small Quantum**: Heavy overhead, but highly responsive system

**Large Quantum**: Minimum overhead, but non-responsive system

Quantum should be large compared to the time required for a context switch, but not too large.

A good rule of thumb is that 80% of the CPU bursts should be shorter than the time quantum.

A quantum of around 20-100ms is often a reasonable compromise.

1. **Which algorithms are guaranteeing starvation free and which are not. SJF has a starvation possibility.**

**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

1. **Comparing different scheduling algorithms**

**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

**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.

**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: Advance knowledge of how long a job will execute, 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

**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 1/2

**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** -

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

1. **Difference between unsafe state and deadlock:?**

Deadlock Unsafe State

Unsafe State Deadlock

1. **Cycles and resource allocation graph: cycle does not imply necessarily deadlock (one resource instance per resource type).**

No cycle No deadlock

If cycle:

* one instance per resource type guarantee deadlock
* multiple instance per resource type possible deadlock

1. **4 necessary conditions for a 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

1. **Strict alternation approach: 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

1. **Deadlock prevention is what we use to attack the necessary conditions**

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

e.g., avoid mutual exclusion

- mutual exclusion not required for shareable resources (read-only)

- spooling can help for some resource types

e.g., 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

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

- low resource utilization, starvation possible

1. **Bankers’ which one is it: prevention? detection? avoidance?**

Avoidance

1. **Spooling is an example of attacking which one of the 4 condition?**

Mutual exclusion

1. **More about necessary conditions and possible attacks on them -> how to prevent**

Ensure system will never enter deadlock state: deadlock prevention, deadlock avoidance

Allow system to enter deadlock and then recover: deadlock detection, recovery from deadlock

Avoid mutual exclusion

- mutual exclusion not required for shareable resources (read-only)

- spooling can help for some resource types

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

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

- low resource utilization, starvation possible

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 the

process is suspended, preempted resources are added to the list of resources for which the

process is waiting, process resumed when it can regain its old and new resources.

- only works if we can save/restore state for the resources preempted

- complicated mechanism, possible starvation, non-optimal resource usage

Avoid circular wait condition

- 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

*Deadlock avoidance* can increase resource utilization if some a priori information is available

1. **Dynamic memory allocation: best/first/next fit, no pros and cons**

*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

1. **Fragmentation (internal and external) memory allocation**

*Internal Fragmentation* - Memory internal to a partition becomes fragmented. (Fixed partitioning)

Leads to low memory utilization if partitions are big

*External Fragmentation* - Memory external to all partitions becomes increasingly fragmented.

(Dynamic partitioning) leading to low memory utilization.

1. **Page replacement algorithms: know FIFO, Optimal (not practical), LRU (approximates optimal), Clock (approximates LRU). Know how CLOCK works**

**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

**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

1. **Virtual memory and know calculations**

If a system has a page size, the **page number and the offset** for an address can the found by:

- Convert address to binary

- The LAST binary digits are the offset

- The rest is the page number

Consider a system with logical address space and page size. The system supports up to

of physical memory.

* Conventional single-level page table:

* Inverted page table:

1. **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

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

1. **Page fault definition**

**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, ...

1. OS loads the missing page from disk
2. OS updates the page table
3. OS resumes the process
4. **Global (stealing pages from other processes) vs. local page replacement algorithms difference. Pros and cons**

**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

1. **Memory allocation algorithms**

*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

1. **iNode 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

1. **Different file allocation methods and if they have fragmentation problems: external or internal or both**

**External Fragmentation** - *Contiguous allocation* has external fragmentation after file deletion

ALL file allocation methods hav**e internal fragmentation** problems, since a block device forces you to

access things in a block.

1. **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

1. **Types of information stored in iNodes (pointers to other iNodes, meta data about the file: size/type/modification time, file name is NOT included) file name is stored in the directory entry**

**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
  + ...

1. **disks scheduling algorithms: FCFS, Shortest seek time first, C-SCAN, C-LOOK… know the differences**
2. **Peterson’s algorithm: implementing mutual exclusion**

**Peterson's algorithm** - A non-blocking technique for implementing mutual exclusion

1. **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

1. **Generic question about Banker’s**

Bankers: general resource-allocation graph algorithm for multiple instances per resource type

Require:

* each process must declare a a max 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

Data Structures (only 2 of Max, Allocation, Need needed, 3rd can be inferred):

* Available[j] = k : instances of resource type
* Max[i, j] = k : instances of resource type may be requested by process
* Allocation[i, j] = k : instances of resource type currently allocated to
* Need[i, j] = k : more instances of resource type may be needed by

Algorithm

1. Initialize temporary vectors:

Work = Available;

Finish[i] = false for i = 0, ..., n-1

1. Find an i such that:

Finish[i] = false && Need[i]Work

IF NO SUCH i, GO TO STEP 4

1. Update Work and Finish:

Work = Work + Allocation[i]

Finish[i] = true

1. If Finish[i] == true for all i

Safe state, return true

else

Unsafe state, return false

1. **Translation lookaside buffer (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:

1. **Calculation about logical vs. physical addresses**

If a system has a page size, the **page number and the offset** for an address can the found by:

- Convert address to binary

- The LAST binary digits are the offset

- The rest is the page number

Consider a system with logical address space and page size. The system supports up to

of physical memory.

* Conventional single-level page table:

* Inverted page table:

1. **Calculate something with page sizes -> similar to assignment**

If a system has a page size, the **page number and the offset** for an address can the found by:

- Convert address to binary

- The LAST binary digits are the offset

- The rest is the page number

Consider a system with logical address space and page size. The system supports up to

of physical memory.

* Conventional single-level page table:

* Inverted page table:

1. **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

1. **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

1. **Calculation about proportional memory allocation scheme: proportional to the process size (bigger the process more memory it gets )**

**Proportional Memory Allocation** - Allocate according to the size of the process. This is a *dynamic*

*allocation scheme*: as degree of multiprogramming and process size changes

1. **Page entry within a page table: 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?

1. **Difference between preemptive and nonpreemptive scheduling**

*Non-Preemptive* - Voluntary context switch only

* Multitasking is possible, but only through cooperation
* Process runs until it does a blocking system call (e.g., I/O), terminates, or voluntarily yields CPU
* **FCFS**

*Preemptive* - Involuntary (as in current process stops involuntarily) context switch

* Usually as a direct or indirect result of an interrupt, but not limited to clock interrupt
  + e.g., new job added, existing process unblocked
* **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
* **RR**

1. **Very last slide of the very last lecture: 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 timers

1. **Fork() 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);

}

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

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

1. **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)

1. **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)

1. **Bash command**

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}''

1. **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)

1. **FAT for calculation: similar to assignment**

**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

1. **Optimal paging algorithm**

**Optimal Paging** - Replaces the 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

1. **Effective memory access time calculation - Know formula**

probability of page fault or page fault rate

memory access time

= page fault service time

**WITH TLB**:

= probability of TLB hit

= TLB search time

1. **Compare scheduling algorithms for the CPU: what are the advantages and disadvantages**

**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

**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.

**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: Advance knowledge of how long a job will execute, 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

**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 1/2

**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** -

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

Long Answer:

1. **File allocation table: give indices of blocks (assignment). content/directory content -> how many blocks? what are the first and last blocks**
2. **Reference string and which algorithm to use -> how many page faults a particular page replacement algorithm will generate**

**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

**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

1. **Calculations for page tables: size, # of entries, maximum physical size, maximum virtual size supported from the page table. Given content, translate one address to the other. (Page number => frame number +offset and know how to do it the other way. Physical -> logical and search the page table for info)**

If a system has a page size, the **page number and the offset** for an address can the found by:

- Convert address to binary

- The LAST binary digits are the offset

- The rest is the page number

Consider a system with logical address space and page size. The system supports up to

of physical memory.

* Conventional single-level page table:

* Inverted page table:

1. **Total number of head movements for a particular set of requests and 2/3 different disk scheduling algorithms. In which order are the requests served**
2. **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

1. **Allocation graph: draw a graph and show if there’s a deadlock and justify**

set of all PROCESSES in the system. use BOX

set of all RESOURCE TYPES in the system. use OVAL

REQUEST EDGE

ASSIGNMENT EDGE

* NO CYCLE NO DEADLOCK
* CYCLE:
  + ONE INSTANCE PER RESOURCE TYPE DEADLOCK
  + OTHERWISE CHECK MANUALLY

1. **Banker’s algorithm: show if something is in safe state or not. whether a particular request can be granted or not**

Bankers: general resource-allocation graph algorithm for multiple instances per resource type

Require:

* each process must declare a a max 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

Data Structures (only 2 of Max, Allocation, Need needed, 3rd can be inferred):

* Available[j] = k : instances of resource type
* Max[i, j] = k : instances of resource type may be requested by process
* Allocation[i, j] = k : instances of resource type currently allocated to
* Need[i, j] = k : more instances of resource type may be needed by

Algorithm

1. Initialize temporary vectors:

Work = Available;

Finish[i] = false for i = 0, ..., n-1

1. Find an i such that:

Finish[i] = false && Need[i]Work

IF NO SUCH i, GO TO STEP 4

1. Update Work and Finish:

Work = Work + Allocation[i]

Finish[i] = true

1. If Finish[i] == true for all i

Safe state, return true

else

Unsafe state, return false

1. **Schedule some processes using shielding algorithms (SJF, RR, FCFS)**

**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

**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.

**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: Advance knowledge of how long a job will execute, 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

**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 1/2

**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** -

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

1. **Multithreaded programming: what’s wrong with this code, describe**