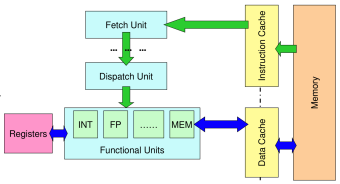
CS2106 Finals Cheat Sheet

Process Abstraction

Computer Organization

- Code snippet from the program is loaded into the memory
- Instructions may be loaded into instruction cache, where the *fetch unit* takes
- Fetch unit then brings the instruction to the dispatch unit, which dispatches it to the correct functional units based on instruction type(memory, arithmetic)



Stack Memory

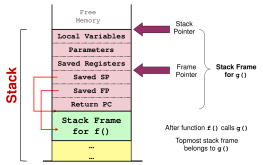
Used by function calls to maintain local variables within the appropriate scope, and to ensure that variables do not outlive it's lifetime/scope

- Each function call is allocated a *stack frame*, which is a portion in the stack memory belonging to the function only that allows it to store it's local data
- Stack frames on placed one atop another in stack memory, with the most recent function call having it's stack frame on the topmost
- When the function call is done, it's stack frame is popped off the top of the stack

Stack Frame Setup

Caller(*f()*):

- Caller passes the arguments needed for the function call(*g()*)
- Caller saves the return PC on top of the stack



Transfer to Callee control

Callee(*g()*):

- Saves the old Stack Pointer(SP) and Frame Pointer(FP) from the Caller onto stack
- Save registers used by Callee if there is any register spilling
  - Register Spilling: if there are not enough General Purpose Registers(GPRs) for use for instructions for the Callee, Callee dictates some registers currently used by another function/program for use in this function call, but 'spills' the contents of these registers which may be intermediate calculations from another function to memory
  - These registers which are now freed can be used for intermediate calculations by the function callee for the duration of it's stack frame, and the spilled contents are restored upon teardown to ensure previous intermediate calculations by another instruction is not lost
- Allocate space on top of stack for local variables
- Adjust the SP and FP to current correct position

Stack Frame Teardown

Callee(*g()*)

- Restore the saved SP, FP, spilled registers and return PC

Transfer control back to Caller

Caller(*f()*):

- Continues execution in caller function

Variables in a stack frame(local variables, arguments) are accessed via an offset from the frame pointer, while stack pointer dictates the start of new free memory

- Frame pointer is necessary because stack pointer is not fixed - may change if variables are allocated mid way through the function call, hence offset cannot be used from this varying position

If there are too many function calls, the stack memory grows indefinitely and eventually running out of space → Stack Overflow

Heap Memory

Dynamically allocated memory that persists outside a function call. Size and lifetime of Heap Memory is dictated by the caller - *Variable size, allocation/deallocation time*

- Heap and Stack memory are not placed contiguously - Space in between them allows for variable allocation depending on demand of memory
- Heap and Stack memory allowed to grow as much as needed without being restricted by free space allocated for the other - prevents internal fragmentation and lack of space produced by fixed allocation scheme

Unix Syscall: *malloc*

```
void *malloc(size_t size);
```

Allocates memory space of specified size in the Heap region, returns a pointer to the start address of the allocated memory if successful, else NULL

- Memory allocated is in terms of raw bytes, the type for the allocated memory can be specified by typecasting the return pointer to the intended type.

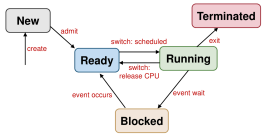
```
void free(void *ptr);
```

Frees the allocated memory space pointed to by the pointer specified

- Needs to be called for each malloc-ed pointer else memory leaks will occur

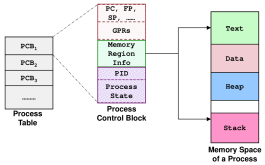
Process States

- Process gets blocked when process goes into I/O, or when process cannot proceed without result from another program
- If sleep() syscall is called, process also goes into blocked state the period of time specified
- Process can never go straight into running from blocked, needs to first go to ready state to join the ready queue and be scheduled by the OS



Process Control Block

- The processor has a process table that keeps track of all current processes. Each process has a process control block (PCB)
- Each PCB contains the following:
  - Hardware context of the process: registers used, PC, SP, FP
  - Memory context: Addresses used by the process
    - For paging scheme: the page table used by the process is placed(single paging), or the page directory (2-level paging)
    - Segmentation scheme: Addresses to each of the segments is specified(If there is not hardware support for registers for segments)
  - Process information: Process ID (PID), parent process ID(PPID), and process state



The process table has limited space, and if there are too many processes being formed, the process table will contain too many PCBs → Unable to create any more process (Consequence of *fork() bomb*)

Unix Syscall:fork()

```
pid_t fork(void);
```

The fork() system call creates a new process with a duplicated copy of the memory context of the parent process(same instruction set, same stage of execution(PC), same variables)

- PID and PPID of the parent and child processes will be different
- fork() returns the value of the child PID to the parent process, and 0 to the child process → useful for segregating parts of code for parent and child to run using control flow
- Copy-On-Write(Using paging scheme): Upon fork(), the page table in the parent process is duplicated into the child process which points to the same frames as the parent
  - If the child process reads from the frame, page continues to point to same frame as parent. If a value in the frame pointed to by both the parent and child is modified by the child, then the values in that frame is duplicated into another frame. The child's corresponding page points to this new frame and it writes to this new frame instead of the one pointed to by the parent.

Unix Syscall: exec()

Family of system calls that allows processes to replace itself with another process

Since the exec() syscall replaces the process calling it with another, to preserve the original process, a fork() + exec() call is done to transform the child process to another intended process(e.g shell interpreter commands)

Unix Syscall: wait()

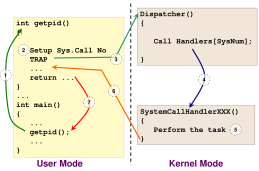
```
pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

The wait() system call causes the calling process to wait for an existing child process, or a child process with a specific PID(waitpid()), if any. The PID of the child process wait-ed on is returned

- wait() syscall is blocking, but non-blocking can be specified for waitpid()
- wait() syscall functions as a form of cleanup for the child process:
  - Upon process termination, a process goes into 'zombie' state: It's status in the PCB is terminated(process has exited), but the PCB still exists in the process table in processor → wait() removes the PCB
  - If the child process exits after parent process calls wait(), parent waits for the child process to exit and cleans up using wait
  - If child process exits after parent, child goes into 'zombie' state. If parent does not call wait() before terminating, then init() process takes over as the parent of the child process and calls wait() on the child process to clean it up.

System Call Mechanism

- Program calls a library call, which can be a function wrapper/adapter for the actual system call.
- Library call puts a system call number in a special register
- OS calls for switch from user to kernel mode using TRAP
- Dispatcher Unit calls the appropriate system call using function pointers with the specified system call number



Process Scheduling

- Execution time for processes divided into intervals → Time Slicing
  - Too short time slices → OS needs to wake up more times, more context switches → higher overhead
  - Too long time slices → Processes allocated too much time → unresponsive

Batch Processing Algorithms

Aimed at maximizing CPU time utilisation for machines that have minimal I/O

First Come First Serve(FCFS) → Non-Preemptive:

Tasks are stored in a FIFO queue, in order of which the task arrives. Task at the front of the queue is picked as the next task to run

- No starvation guaranteed → waiting time for a task X is always bounded by the number of tasks in front of it in the queue
- Not very good turnaround time → total waiting time can be improved by rearranging the tasks in the queue
- Convoy effect: All tasks may be blocked on CPU/ I/O queue at once while the other queue is not being utilized → tasks spend most of it's time blocked

Shortest Job First(SJF) → Non-Preemptive:

Job with the shortest time to execute will be selected to run

- Guarantees shortest waiting time possible for the set of tasks
- Starvation for jobs with long execution time, especially if short jobs keep arriving → Biased towards short jobs as jobs with short execution time will be picked to run each time

Shortest Remaining Time(SRT) → Non - Preemptive/ Preemptive:

Variant of the SJF algorithm, but each interval the OS wakes up, it picked the job with shortest remaining time instead of shortest total runtime.

- Preemptive version means that short jobs that arrive that pre-empt currently running long jobs

- Provides good service for short jobs, even if they arrive late
- Starvation for long-running jobs is now worse, as short jobs that arrive that pre-empt currently running long jobs

- Starvation problem for SJF and SRT can be solved using an active set and expired set. Jobs are picked from the active set based on SJF/SRT algorithm, and placed in the expired set if they are not finished. Tasks in the active set must first be finished before the expired set is swapped with the active set, ensuring the wait time for long running jobs is bounded

Interactive Environment Algorithms

Interval of Timer Interrupt(ITI) → Interval which the OS is triggered to perform scheduling

Time Quantum(TQ) → Time period given for a program to run each time

- TQ should be a multiple of the ITI → to ensure consistency in time periods
  - If TQ ≠ kITI, then if a program gives up CPU early, there is a period of time where CPU is completely idle. If program runs through it's entire TQ, at the end of the TQ the scheduler does not wake up and program can overrun it's TQ

Round Robin(RR) - Pre-emptive

Scheduler runs tasks in a given set in a fixed cyclical fashion → pre-emptive version n of FCFS

- Guaranteed no starvation → waiting time bounded by the tasks before it
- Choice of TQ is important: Long TQ → longer running time(greater CPU utilization) per process, but less responsive. Short TQ → greater number of interrupts → higher overhead incurred for more context switches

Priority Scheduling - Pre-emptive/ Non Pre-emptive

Each task is assigned a priority value, and each time the task with highest priority selected to run

- Starvation for low priority tasks, especially for pre-emptive version
  - Priority for running tasks can be decreased the each time the process runs, eventually allowing lower priority tasks to have a higher priority than initially high priority tasks and run
- Priority inversion: Lower priority task locks a resource due to being pre-empted by a higher priority task. High priority task needs to share the same resource as the low priority task which is locked, and hence cannot run → blocked, allowing the next task with lower priority than it to run instead

Multi-Level Feedback Queue(MLFQ) - Non Pre-emptive

Basic Rules:

1. If Priority(A) > Priority(B), then A runs
2. If Priority(A) = Priority(B), A and B run in round robin fashion

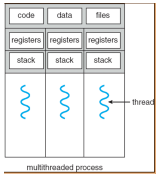
Priority Rules:

1. New task that arrives is assign the highest priority
2. If task fully utilises it's TQ, it's priority is lowered
   - CPU-bound processes that fully utilizes it's TQ will eventually sink in priority
3. If task gives up it's TQ before it finishes → priority is retained
   - Allows I/O bound processes to be responsive → I/O processes will block for I/O → give up the CPU early during this stage, and hence retaining it's high priority and allowing I/O processes to be responsive
   - Can be exploited to hog the CPU → processes can be programmed to give up the CPU early just before the TQ expires, hence maintaining high priority and hogging the CPU over other tasks

- Adaptive → Learns the behaviour of the processes it schedules

Threading

- Multiple threads shared the same data, text and heap region of memory, but have a separate stack memory
- Good for running processes that requires the same set of instructions and data to be shared



Benefits over Multi-Processing

- No context switch required, only need to switch between stack memory of different threads → can appear much more responsive
- Can exploit multiple CPUs (for multiple kernel threads)
- Easier to manage due to less overall resources/overhead →

User Threads

Threads implemented as library calls in this mode

- Flexible → can access user threads as long as the library exists, not hardware dependent
- OS is not aware of these user threads → user threads cannot exploit multiple CPUs
  - All user threads running seen as a single process to the OS single thread blocked means entire process blocked to the OS → all threads blocked

Kernel Threads

Threads are now part of the system construct

- OS can allocate multiple kernel threads to a single process, allowing multiple threads to run at the same time for a single process → can exploit multiple CPU
- Kernel threads now system call → more resource intensive
- Hardware dependent, thread features may vary from system to system
  - Kernel threads used for all types of processes, hence sufficient features need to be implemented for kernel threads → resource intensive

Hybrid Model

- Single process may have multiple user threads and multiple kernel threads
- User threads are bound to kernel threads on which they run m
  - If multiple user threads bound to single kernel thread → user threads switch between each other on scheduling basis
  - Each kernel thread schedules it's bound user threads separately from other kernel threads

Unix Syscall: pthread

```
int pthread_create(pthread_t* tidCreated, const pthread_attr_t* threadAttributes, void* (*startRoutine) (void*), void* argForStartRoutine);
```

tidcreated - thread ID to be set for this thread

threadAttributes - attributes for the thread

startRoutine - Function to be passed for the thread to execute

- Function must return a void* and take in a single argument of type void*

argForStartRoutine - void* type argument to be passed in to the function executed by the thread

```
int pthread_exit( void* exitValue );
```

Library call that exits from this thread, regardless of control flow in current thread

- return call in the routine function is an implicit call to pthread_exit the return value as the exit value, but return calls in a thread do not always imply pthread_exit()
- Exit value defines the exit value to be returned to the caller

```
int pthread_join( pthread_t threadID, void **status );
```

Waits for the termination of the thread specified with ID threadID

- Status defines the exit value/return value of the terminated thread
  - Return value of 0 dictates success by default

Inter-Process Communication

Shared Memory

A region in memory is allocated for use for multiple processes

1. Process A first creates a shared memory region, and attaches the shared memory region to itself
2. Process B then attaches the same shared memory region to itself. Both processes now access the same shared memory region

- Using paging scheme, respective shared memory pages in each process points to the same frames in memory
- Efficient for OS to allocate → OS only needs to handle creation, allocation and attaching of shared memory regions to the respective processes. Reading and writing of data to the shared memory is independent of OS
- Shared memory region treated as though it is a normal memory region belonging to the process → Easy for processes to use
- Synchronization is required → interleaving read and write operations by multiple processes could cause race conditions

Unix Syscall: shmget()/shmat()

```
int shmget(key_t key, size_t size, int shmflg);
```

shmget system call creates a new shared memory region of the specified size, and returns the ID associated with the created shared memory region, or -1 if creation of new shared memory region is unsuccessful

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

shmat system call attaches the shared memory associated with the ID shmid to the current process. shmaddr specifies the exact logical address at which to attach the shared memory segment to the process. Returns the memory address of the attached memory address, or -1 on error

```
int shmdt(const void *shmaddr)
```

shmdt detaches the shared memory region with the given shmaddr, usually specified by the shmat call. Returns 0 on success, and -1 on error.

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

shmctl performs control operations on the shared memory region given by ID shmid. If IPC_RMID is specified for cmd argument, the memory region is destroyed.

Message Passing

Processes communicate in the form of passing messages in order to shared data

- Can be direct → messages explicitly state the recipient of the message, or indirect → messages sent by process goes to a port shared by processes
- Portable→ can be easily implemented on multiple OSes
- No synchronisation issue → no direct sharing of data
- Higher overhead incurred by the OS → need to manage the messages between processes
- Limited size and format of messages restrict flexbility of operations

Unix Pipes

- Functions as a form of duplex buffered channel between processes
- Output from one process can be piped to another via pipe → Writing process waits when pipe if full, reading process waits if pipe is empty

Unix Signals

A form a communication between processes regarding an event.

- Unlike interrupts, which are form of communication between the CPU and the OS kernel(e.g scheduling every given ITI, TRAP), signals are sent as an asynchronous notification between processes and the OS kernel

```
sighandler_t signal(int signum, sighandler_t handler);
```

signal system call can be used to specify a custom signs handler for the signal with ID signum. The specified signal handler must be of signature void (*sighandler_t)(int);

- Signals SIGKILL and SIGSTOP cannot be intercepted in this manner

Synchronization

Race condition - Interleaving instructions of operations from two different programs sharing the same resource may result in non-deterministic result depending on the order of the interleave → need to protect segment of code that accesses shared resources(Critical Section)

Time	Value of X	P1	P2
1	345	load X → Reg1	
2	345	Add 1000 to Reg1	
3	345		Load X → Reg1'
4	345		Add 1000 to Reg1'
5	1345	Store Reg1 → X	
6	1345		Store Reg1' → X

Principles of Synchronization

Mutual Exclusion - If a process is in CS, then all other processes cannot enter

Progress - If no process is in CS, one of the waiting processes should be granted access

Bounded Wait - There exists an upper bound number of times other processes can enter CS before current process can enter again

Independence - Process not in CS should not block other processes from entering

Test and Set - Low Level Synchronization

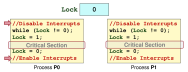
1. When data in a memory location needs to be accessed, it's content is loaded into a register.

2. The value 1 is then put into the memory location → 1 indicates the memory location is locked

- Locks the resource, but other processes who require the resource continually waits on the resource until released → Busy waiting, waste of processing power

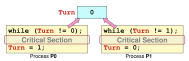
Peterson's Algorithm - High Level Implementation

Attempt 1: Using only Lock object



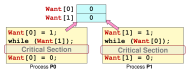
- Mutual exclusion not satisfied → interleaving the line while(lock != 0) for both processes allows both processes to satisfy the condition at the same time and enter CS
- Disabling interrupts to prevent context switch solves the problem, but buggy CS may cause whole system to be stalled → cannot be interrupted

Attempt 2: Using Turn instead of Lock



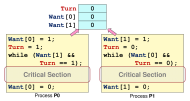
- Violates Independence → If process 1 doesn't enter the CS, process 2 cannot enter

Attempt 3: Using Want instead of Turn



- Independence problem is resolve but causes deadlock → if want[i] = 1 is interleaved for each process I

Peterson's Algorithm



- Using turn and want together guarantees mutual exclusion, independence and no deadlock
  - If other process does not request to go into CS(want[1] == 0), process can proceed into CS without waiting
  - If both processes want to enter CS, one process entered depending on value of turn → no deadlock
- Busy waiting still prevalent
- Specific mechanics rather than general → cannot be used in all scenarios

Semaphore - High Level Abstraction

Synchronization Object that contains two operations:

Wait(Semaphore) - Decrements Semaphore if Semaphore is > 0, else if Semaphore ≤ 0, goes into blocked state

- Placed before the CS to indicate if a process is in CS(Semaphore = 0), or CS is free(Semaphore = 1)
- For a general semaphore, Semaphore value indicates how many processes can enter the CS at once.

Signal(Semaphore) - Increases the value of Semaphore by 1. If there are any sleeping processes waiting on the Semaphore, wake one of them up

- This operation is non-blocking

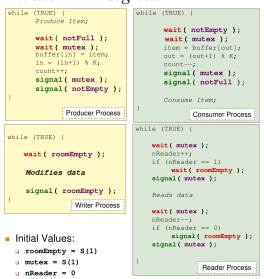
Barrier - Semaphore that proceeds only if all processes have entered

```
Barrier( N ) {
    wait( mutex );
    arrived ++;
    signal( mutex );

    if (arrived == N )
        signal( waitQ )

    wait( waitQ );
    signal( waitQ );
}
```

- First semaphore mutex to prevent race condition when updating number of processes arrived
- Second semaphore blocked at 0 until the last process enters → signal to 1
- Processes that proceed through the barrier waitQ need to free the semaphore afterwards by calling signal on it

Invariant for Semaphore: $S_{current} = S_{initial} - N_{wait} + N_{signal}$

Producer-Consumer



- Both producer and consumer access the items in a shared buffer → needs to be locked using mutex
- Semaphores notFull/ notEmpty used to signal Producers/Consumers to produce/consume → prevents busy waiting
  - notFull initialised to number of Produces, notEmpty initialised to 0

Reader-Writer

- Multiple readers can read from the buffer at once, since no change to the data being made.
  - If first reader (nReaders == 1) → signal to writers that data being read (wait(roomEmpty))
  - If no more readers in room after reading (nReaders == 0) → signal to writers that room empty, can now begin writing (signal(roomEmpty))
- Only one writer can modify the data at once → wait(roomEmpty), and only when no reading is taking place
- Heavily biased towards readers → multiple readers can continually enter the CS
  - Can be resolved using an additional semaphore revDoor → indicates that writer wants to write → readers must then wait(revDoor) to see if any writer wants to write

Dining Philosophers

- Deadlock due to symmetry of each process sharing resources in a cyclical fashion
  - Deadlock can be broken by removing the symmetry:
    - Removing one of the processes from participation at once → max number of processes participating is N - 1 → *Limited Eater Solution*
    - Letting one of the processes be a 'left hander' and the others 'right handed'

Tanenbaum Solution

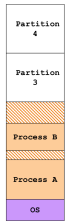
```
void putChpSticks( i )
{
    wait( mutex );

    state[i] = THINKING;
    safeToEat( LEFT );
    safeToEat( RIGHT );

    signal( mutex );
}

void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

        state[ i ] = EATING;
        signal( s[i] );
    }
}

void takeChpSticks( i )
{
    wait( mutex );
    state[i] = HUNGRY;
    safeToEat( i );
    signal( mutex );
    wait( s[i] );
}
```

- Process checks if left and right processes are in the CS → signals it's own semaphore if can enter
- Else if left and/or right processes in CS, need to wait for them to signal current process to enter CS

Contiguous Memory Schemes

Assumption: Physical memory is able to contain one or more processes with complete memory space, and each process takes up contiguous memory

Fixed Size Partitioning



Physical memory is divided into partitions of fixed size → Size of each partition should be that required by the largest process

- Easy to manage and allocate → Each partition is fixed size, hence fixed offset is used to locate free partitions.
- Internal fragmentation → Since partition size allocated for the largest process, any process that uses less memory than the largest will waste the remaining memory space in the partition

Dynamic Size Partitioning

Memory space as needed by the process is allocated exactly. Linked List of allocated and free spaces are maintained.

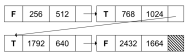
- Removes internal fragmentation → flexible in allocation
- Introduces 'holes' of free space in between allocated spaces as memory spaces are freed → External fragmentation
  - Can be solved by merging nearby free spaces (Merging) and shifting free memory spaces together to consolidate and merge them (Compaction)
- OS needs to maintain and keep track of free and allocated spaces → higher overhead

On finding a free block of size M bytes, process is allocated N bytes from the M bytes and a new free block of M - N bytes is formed. Three allocation schemes:

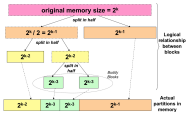
- First Fit: Allocate the first free space that is large enough
- Best Fit: Allocate the smallest free space that is large enough
  - May accumulate many small, but unstable free blocks without merging
- Worst Fit: Allocate the largest free space that is large enough
  - Leftover memory space from the block not used after allocation may still be large enough for another process to use

For each node in the linked list, following maintained:

1. Free/Allocated bit 2. Start sector 3. Block Space



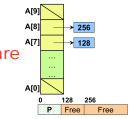
Buddy System



Entire physical memory is segmented into sizes of power 2

Allocation:

1. Array of size [0…k] is maintained where 2^k is the largest allocatable block size
   - 1.1. At each index *i*, a linked list of the starting addresses of free blocks of size 2^i is maintained → if there are no free blocks left, array will be empty
   - 1.2. Initially, only a single block of size 2^k is present



2. Upon allocation of memory block of size N, the smallest integer S is found such that $2^S \geq N$, and the linked list at index S in the array is searched
   - 2.1. If a free block is found, the block is removed from the linked list and allocated to the process using the starting address
   - 2.2. If no free blocks are found, *R* is found by traversing from *S + 1* to *K* up the array for a free block. The free block is then repeatedly split in two until index *S* in the array has a free block.

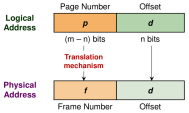
Deallocation:

1. For a block of size 2^S to be freed, the block is first inserted into the linked list at index *S* of the array
2. The linked list of the array at index *S* is checked if the buddy of the block is found.
   - 2.1. Buddy blocks are two consecutive blocks are originate from the same parent after splitting. For a block B with starting address $X \ldots X0 \ldots 0_2$ / A the buddy block B' has $X \ldots X1 \ldots 0_2$ / $A + 2^S$
3. If buddy blocks are found, the two blocks are removed from the linked list at index *S* of the array and merged, creating a new block of size 2^{S+1} which is inserted into the index *S + 1* in the array
4. Merging is recursively repeated if buddy block for new free block inserted is found

Disjoint Memory Schemes

Paging Scheme

- Physical memory split into small units known as physical frames
- Logical memory is split into small units of the same size as the physical frames known as logical pages → one to one mapping of pages to frames
- Each process has a page table in it's memory context that stores the mappings of pages to frames



- Each logical address has:
  - Page Number: used to obtain the mapping to the corresponding frame number
  - Offset: To obtain the specific data sector within a frame, an offset is taken from the start of the mapped physical frame
- Frame size should be power of 2 → Easy obtaining of physical address which is given by frame number *F* x size of physical frame + offset by bit shift
- Removes external fragmentation due to disjoint scheme, but may still have internal fragmentation is frames are not fully utilized

Translation Look-Aside Buffer → Caching of page table entries:

- Without TLB → need to access page table in memory → *2 x* memory access time
- With TLB → first access TLB, if page not present in TLB then access page table → *TLB Hit + TLB Miss time* → (cache access time + memory access time(to get physical frame)) x $\%_{hit}$ + (2 x memory access time) x $\%_{miss}$
- When page needs to be accessed → process looks into TLB to get frame mapping → if not present, OS brings pages in same locality as requested page
- On context switch, TLB flushed → prevent other processes from seeking previous TLB
- Protection: Permission bits, Access Right bits(read/write/execute)

Segmentation Scheme

- Text, Data, Heap and Stack memory region separated into contiguous segments
- Each segment is represented by an ID, and has a base address and a limit, and logical address given by <SegID, Offset>

- SegID used to look up the base address in the segment table → offset first compared with limit, if offset > limit → segmentation fault

Hardware Support → since number of memory regions is small, dedicated memory can be allocated to 'cache' the segment table

- Each segment a contiguous memory region → able to grow and shrink as needed
- Contiguous dynamic size allocation → can cause external fragmentation

Segmentation with Paging

- Process memory is divided into segments using segmentation scheme, and then each memory segment is divided into pages
  - Logical address given as <SegID, Page Number + Offset> where Page Number + Offset is a X + Y bit number, where the first X bits is Page Number and the next Y bits is the Offset from the mapped frame →Offset determined by page size, number of bits to represent page is number of bits of offset
  - Solves external fragmentation introduced by segmentation using pages
  - Size of segment limited to the page table size/ number of bits in page number

Virtual Memory Scheme

Used when the memory required by the process is larger than the memory in CPU

- Total number of pages > number of frames in memory → if page is not required, it is shifted to secondary storage (Non-memory resident)
  - Accessing page X → first check if page X is in the corresponding page table
  - If not in page table → TRAP to OS, OS locates the intended page in secondary storage and loads into existing frame → update page table
  - OS brings multiple frames in vicinity of intended frame to page table → secondary storage access is very slow, exploit spatial locality

2 - Level Paging

Due to large number of logical pages, page table too large to store all pages

- Single page directory used → each index in the page directory maps to a single page table → page tables allocated only when needed
- Processes now store the page directory, and page tables allocated are stored in memory and pointed to when the process's page directory is loaded → page tables should be size of one frame to avoid spilling into other frames in memory
  - Since page table in frames → can be shifted to secondary storage when not needed → page faults can occur at two levels: page table fault + page fault
- Allows more processes to be run without being limited by the processor memory

Inverted Table → reverse mapping from frame number to <PID, Page Number>

- Allows OS to identify which frame is used by what process without looking through page tables for all processes

Memory Access Time: $T_{access} = T_{mem} \times (1-p) \; + \; T_{page\ fault} \times p$

- T_{mem} = *2 x time taken to access memory*
- $T_{page\ fault}$ = *2 x time taken to access memory + time taken to access SS*

Page Replacement Algorithms

Optimal (OPT) → Replace page that will not be used for the longest time

- Needs future knowledge of memory references → not implementable
- Serves as a benchmark for other algorithms → OPT guarantees minimum number of page faults → closest to OPT as possible is ideal

FIFO → Memory pages evicted based on loading time, earliest loaded evicted first

- Each time memory page loaded into frame, OS adds loaded page into a queue → first page in queue evicted when needed(oldest page loaded)
- Easy to implement → no hardware support required
- As page number increases, number of page faults increases → Belady's Anomaly
  - This is because FIFO does not exploit temporary locality

Least Recently Used → Replace page that has not been used for the longest time

- Uses assumption of temporal locality → page that is recently used will be likely to be used again and converse is true
- Difficult to implement → using a counter may cause overflow, and implementing a stack is difficult in hardware

Second Chance → Pages are given second chance if accessed recently

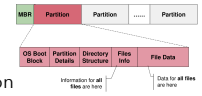
Modified FIFO algorithm that introduces a 'useful' bit to each page added

- On page replacement, oldest page selected from the queue
  - If reference bit is 0 → page is replaced
  - If reference bit is 1 → second chance given to the page. Reference bit for selected page set to 0, and next oldest page is selected
  - Degenerates into FIFO algorithm if all bits are the same (0/1)

Frame Allocation

Working Set Model

- A working set window $W(t, \delta)$ defines the number of frames used by a process in a time frame of t, where δ represents the number of frames used in the interval
  - Local replacement → Frames replaced are taken from within the process only.
    - Stable performance across all the processes, but if frames initially not enough, entire process may be slow overall
  - Global replacement → Frames may be taken from other processes
    - Processes can self adjust for increased performance, but bad behaving processes can steal frames and hog CPU



File Systems Implementation

File Allocation

File Data segregated into data blocks → internal fragmentation always a prevalent problem if block not fully utilized

Contiguous File Allocation → Files are assigned contiguous regions of memory

- Easy to allocate and maintain → only the start block and length needs to be maintained for access
- External fragmentation → Problem of contiguous dynamic allocation

Linked List → File data consists of disjoint blocks in a linked chain

- Each data block contains file data, and also a pointer to the next block
  - Last block contains a special value to indicate end of file data
  - Since pointer contained inside the block → some of the space in the block needs to be allocated for pointer before file data → file data cannot utilise all the space in the block
- Slow random access → traversal of linked list to get specific block is slow in secondary storage

Indexed Allocation → Using disjoint blocks with an array of indexed block addresses

File uses a sequence of disjoint blocks, but instead of maintain a pointer to the next block in each data block, file is allocated a data block which stores can array of disk block addresses

- Data block containing the disk block address array is stored in Files Data. array[N] → gives the disk block address for the N^{th} block
  - Fast random access → N^{th} block can be quickly accessed by indexing disk block array → solves slow random access of linked list implementation
  - File size limited by the number of indexes in the array which determines the total number of data blocks → size of data block
    - Can be resolved using linked list of data blocks for disk block array, or multi level indexing

Free Space Management

Bitmap → array of bits that indicate if a block is free/occupied

- Array of bits where bitmap[i] indicates 1 if the i^{th} disk block is empty, else 0
  - Needs to have an index for each disk block → number of indexes/size of array determines the number of data blocks
- Highly efficient operation → but needs to be in memory

Linked List → List of free blocks maintained

- A linked list of free data blocks is maintained, where each data block stores a sequence of free blocks, and also a pointer to the next data block in the linked list
  - Free data block used in storing sequence of free data blocks has it's own free block number at the start of the list → own block allocated last in it's sequence → if own block not allocated last, sequence of free blocks in own block will be overridden by allocated data

Directory Management

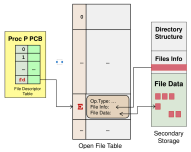
Linear List implementation → Files need to be traversed linearly to search

- Directory structure can be modified into a tree, or can cache file entries

Hash Table implementation → File names hashed and stored in an array using hash as index

- Fast look up if hash function is good

Open/Create Files



- On create/opening of files, absolute file path is used to traverse and locate file
  - Create: If file is found →duplicate, terminates with error. Else, free space allocated for file, and file entry added to parent directory
  - Open: If file not found → terminates with error. Else, located file information loaded into system wide file table → new file descriptor created in process file descriptor table that points to this new file table entry
- Multiple processes open same file → different file descriptors pointing to same file → does not change the file pointer for other processes
  - To share the same file pioneer → file descriptor point to same entry in file table e.g child processes and parent process from fork()

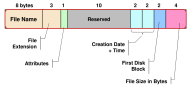
Disk Scheduling → Reduce Seek Time

- FCFS/SSF(Shortest Seek First) → Modified version of FCFS/SJF
- C-SCAN(1-directional), SCAN(bi-directional) → services all sectors in the correction direction. C-SCAN restarts from the smallest, SCAN reverses direction

File Allocation Table(FAT)

- File data are kept in data blocks in a linked list fashion
- FAT table is an array containing block number of next block of block *i*
  - Special codes for EOF/BAD/FREE to indicate the block is free/corrupted/last block → blocks with same block numbers as these codes cannot be used

Directory



Contains a list of directory entries each with:

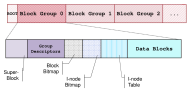
- File Name → In 8+3 format(8 char for name, 3 for extension)
  - Virtual FAT(VFAT) supports longer file names by using additional directory entries → uses invalid attribute bits to prevent reading
- Start sector → for getting the first disk block of file. Number of bytes specify FAT type(2 bytes → 2^{16} blocks → FAT16, 4 bytes →2^{32} blocks→FAT32)

Free Space Management → No additional overhead, need to traverse the FAT table

Delete → Replaces the first character of file name with 0xE5. Traverses the FAT table to change linked sector chain blocks to FREE

- can attempt to undo delete, but not guaranteed to recover whole file

Extended-2



File System is segregated into block groups

Partition Information:

Superblock→describes whole file system

- Number of INodes/Disk blocks per block group

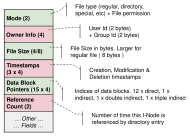
Group Descriptors→describes each block group

- All group descriptors duplicated in each block group for redundancy

INode/Disk Block Bitmap → Keep track of free INodes/Disk Blocks

INode Table → array of INodes that store file information

File Data → stored in INodes



INodes contain the file metadata and pointers to data blocks containing the actual file data

- 12 Direct block pointers → directly points to data blocks
- 1 indirect, double direct, and triple indirect blocks
- Provide fast access to small files and can cater to large files

Directory → Linked List of directory entries

- Each directory entry stores the corresponding INode for each entry, and type(File/Subdirectory)

Hard/Symbolic Links → Sharing Files

Hard Link → File directly points to another file in the FS

- May cause NULL pointers if deletion happens → one file points to nothing
- For ext-2, two directory entries can point to the same INode for same while

Symbolic Link → File is an alias for another file in FS → points to another special block that points to actual file path → if deleted, special block not usable

- For ext-2, symbolic link directory entries can point to INode that stores data block containing absolute path of file → changes in filepath can invalidate symbolic link

Directory Write Access → Can modify Directory Linked List(create new files)

Directory Read Access → Can access Directory Linked List(ls command)

Directory Exec Access → Can access files in directory and directory(cd, vim)

- Can access files in directory with exec permission but without write, as modifying files does not affect modification of directory linked list