CS2106 Finals
AY23/24 SEM 2
github/SelwynAng

1 Synchronisation

1.1 Race Conditions

- Happens when execution of concurrent processes is non-deterministic
- Process which reaches the end first is the one that loses
- Each statement of a process consists of
  1. Load operation (Loads value from resource intro register)
  2. Register operation
  3. Store operation (Stores value from register into resource)
- Bad behavior occurs when different processes interleave
- No. of inter-leavings = $(nm!)/(m!)^n$ where there are n threads, m instructions per thread
- Relevant stages: Load & Store (eg. Thread A has 2 instructions, Thread B has 1 instruction $\longrightarrow$ Thread A has 1L,1S,1L,1S, Thread B has 1L, 1S $\longrightarrow$ Insert B's L into 5 possible slots, insert B's S into 6 possible slots, but since only half of the permutations has L before S $\longrightarrow$ No. of interleavings = 5*6/2 = 15)

1.2 Critical Section

- **Definition of Critical Section:** Section of code that performs a read, update & write of a shared resource (code segment with race condition) $\longrightarrow$ Only 1 process can execute in Critical Section at any point of time
- **Properties of Correct Critical Section Implementation**
  1. Mutual Exclusion: If a process is executing in CS, all other processes are prevented from entering CS
  2. Progress: If no process in CS, 1 of waiting processes should be granted access
  3. Bounded Wait: After a process P_i requests to enter CS, there exists an upper-bound no. of times other processes can enter CS before P_i
  4. Independence: Process not executing in CS should never block other processes
- **Symptoms of Incorrect Synchronisation**
  1. Deadlock: All processes are blocked $\longrightarrow$ No Progress
  2. Livelock: Processes keep changing state to avoid deadlock (processes are not considered blocked) $\longrightarrow$ No Progress
  3. Starvation: Some processes are blocked forever

1.3 Assembly-level Implementation of CS

- Mechanism provided by processor

1.3.1 Test & Set

- **Overview:** Takes a memory address M $\longrightarrow$ Returns current content at M & Sets content of M to 1

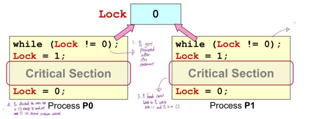
```
void EnterCS( int* Lock )
{
   while( TestAndSet( Lock ) == 1);
}
```

```
void ExitCS( int* Lock )
{
   *Lock = 0;
}
```

- **Procedure:**
  1. Initially, Lock == 0 $\longrightarrow$ `TestAndSet(Lock)` returns 0 and sets Lock == 1
  2. While loop of 1st process exits $\longrightarrow$ 1st process entered CS successfully
  3. Other processes cannot enter CS as Lock == 1 $\longrightarrow$ While loops will not exit for those processes
  4. Upon exiting CS, set Lock == 0 $\longrightarrow$ Other processes can enter CS
- **Cons:** Employs busy waiting for blocked processes (keeps checking while loop condition until it is safe to enter CS) $\longrightarrow$ Waste of processing power

1.4 High-level Language Implementation of CS

- Utilize only normal programming constructs
- **Flaws of certain implementations:**
  1. Preemption: P_0 gets preempted after while loop $\longrightarrow$ P_1 sees that lock == 0, exits while loop & sets lock == 1 $\longrightarrow$ P_1 hands control back to P_0 while lock == 1 & P_1 is in CS $\longrightarrow$ P_0 has checked the while loop before already & would just enter CS (Mutual Exclusion violated)

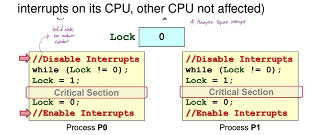


Lock
0
1. P0 gets preempted after this statement
3. P1 hands control back to P0 while lock=1 and P1 is in CS
4. P0 checked the while loop in (1) already, so would just enter CS ⟹ Mutual exclusion violated.

```
while (Lock != 0);
Lock = 1;
Critical Section
Lock = 0;
```
Process **P0**

```
while (Lock != 0);
Lock = 1;
Critical Section
Lock = 0;
```
Process **P1**

  2. Disabling & enabling interrupts: Prevents context switch when a process is in CS (preemption requires interrupts), but
     (1): Process crashes in CS, no way to re-enable interrupts $\longrightarrow$ system shuts down; (2): Busy waiting employed; (3): Permission needed to disable/enable interrupts; (4): Ineffective on multi-CPU systems (If processes on different CPUs, disabling a process' interrupts will only disable interrupts on its CPU, other CPU not affected)

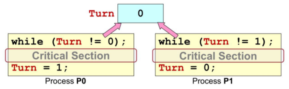


Lock
0
Kind of makes lock mechanism redundant
* Preemption requires interrupts

```
//Disable Interrupts
while (Lock != 0);
Lock = 1;
Critical Section
Lock = 0;
//Enable Interrupts
```
Process **P0**

```
//Disable Interrupts
while (Lock != 0);
Lock = 1;
Critical Section
Lock = 0;
//Enable Interrupts
```
Process **P1**

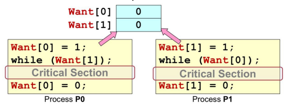
  3. Failure to enter CS: P_0 does not manage to enter CS, P_1 can starve $\longrightarrow$ fufils Mutual Exclusion, but violates Independence



```
while (Turn != 0);
Critical Section
Turn = 1;
```
Process **P0**

```
while (Turn != 1);
Critical Section
Turn = 0;
```
Process **P1**

  4. Deadlock: P_0 sets `Want[0] == 1` & preempted $\longrightarrow$ P_1 sets `Want[1] == 1` $\longrightarrow$ Both processes are stuck at while loop

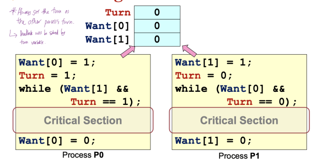


```
Want[0] = 1;
while (Want[1]);
Critical Section
Want[0] = 0;
```
Process **P0**

```
Want[1] = 1;
while (Want[0]);
Critical Section
Want[1] = 0;
```
Process **P1**

1.4.1 Peterson Algorithm

- **Overview (Analogy of boarding bus where bus is CS):** A process will WANT to enter CS & then gives TURN to other process $\longrightarrow$ If P_i gives TURN to P_j but P_j don't want to enter CS, P_i will enter CS $\longrightarrow$ Final TURN value ultimately determines which process can enter CS (only when both processes WANT to enter CS)



* Always set the turn as the other process's turn.
↳ Deadlock will be solved by turn variable.
Turn
0
Want[0]
0
Want[1]
0

```
Want[0] = 1;
Turn = 1;
while (Want[1] &&
       Turn == 1);
Critical Section
Want[0] = 0;
```
Process **P0**

```
Want[1] = 1;
Turn = 0;
while (Want[0] &&
       Turn == 0);
Critical Section
Want[1] = 0;
```
Process **P1**

- **Scenarios:**
  1. Both P_0 & P_1 WANT, TURN == 0 $\longrightarrow$ P_0 enters CS, P_1 blocked
  2. Both P_0 & P_1 WANT, TURN == 1 $\longrightarrow$ P_0 blocked, P_1 enters CS
  3. P_0 WANT & P_1 don't WANT, TURN == 1 $\longrightarrow$ P_0 enters CS, P_1 will get blocked even if it WANTS & sets TURN == 0 eventually
  4. P_0 don't WANT & P_1 WANT, TURN == 0 $\longrightarrow$ P_1 enters CS, P_0 will get blocked even if it WANTS & sets TURN == 1 eventually

1.5 Semaphores Implementation of CS

1.5.1 Semaphores

- Provides a way to block a number of processes & a way to unblock $\geq$ 1 sleeping process
- A semaphore `S` contains an integer value (Initialised to any non-negative value)
- Given $S_{initial} \geq 0$, $S_{current} = S_{initial}$ + No. of signal operations executed - No. of wait operations completed
- **Semaphore Operations:**
  1. `Wait(S)`: If `S` $\leq$ 0, process blocks $\longrightarrow$ After process unblocks, decrement `S` (Code: `P(Semaphore S) { while(S <= 0); S --;})`
  2. `Signal(S)`: Increment `S` & unblocks 1 blocked process if any (Signal never blocks) (Code: `V(Semaphore S) {S++;})`
- **Semaphore Types:**
  1. General Semaphore: S $\geq$ 0
  2. Binary Semaphore: S = 0 or 1 (General semaphores can be mimicked by binary semaphores)
- **Semaphores in Critical Section:**
  - Place `Wait(S)` before CS & `Signal(S)` after CS $\longrightarrow$ ensures mutual exclusion
  - Binary Semaphore Implementation:(1): $S_{initial}$ = 1 & there are 2 processes P_0 & P_1 trying to access shared resource in CS; (2): P_0 calls `Wait(S)` $\longrightarrow$ Since S = 1, it does not block & proceeds to decrement S to 0 $\longrightarrow$ P_0 enters CS; (3): P_1 calls `Wait(S)` $\longrightarrow$ Since S = 0, P_1 blocks & cannot enter CS; (4): P_0 finishes CS & calls `Signal(S)` $\longrightarrow$ S incremented back to 1; (5): P_1 becomes unblocked, S decremented back to 0, P_1 enters CS, calls `Signal(S)` & increments S to 1
  - General Semaphore Implementation: Can be generalised from binary semaphore implementation (eg. $S_{initial}$ = 2, 3 processes trying to access 2 shared resources in CS)

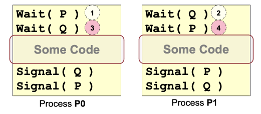
1.5.2 Proofs of Semaphore Correctness

1. **Semaphore ensuring mutex in CS:** N_{CS} = No. of processes in CS = #Wait(S) - #Signal(S) $\longrightarrow$ Given $S_{initial}$ = 1, $S_{current}$ = 1 + #Signal(S) - #Wait(S) $\longrightarrow$ $S_{current} + N_{CS} = 1$ $\longrightarrow$ Since $S_{current} \geq 0$, hence $N_{CS} \leq 1$
2. **Semaphore preventing deadlock:** Deadlock means all processes stuck at `Wait(S)` $\longrightarrow$ $S_{current} = 0$ & $N_{CS} = 0$ $\longrightarrow$ But $S_{current} + N_{CS} = 1$ (Contradiction)
3. **Semaphore preventing starvation:** Suppose P_1 is blocked at `Wait(S)`, P_2 is in CS $\longrightarrow$ P_2 exits CS with `Signal(S)` $\longrightarrow$ If no other processes sleeping, P_1 wakes up OR If there are other processes, P_1 eventually wakes up (assuming fair scheduling)

1.5.3 Disadvantages of Semaphores

- Can result in Deadlock if used incorrectly

Assume semaphores **P = 1**, **Q = 1** initially



```
Wait( P )
Wait( Q )
Some Code
Signal( Q )
Signal( P )
```
Process **P0**

```
Wait( Q )
Wait( P )
Some Code
Signal( P )
Signal( Q )
```
Process **P1**

1. P_0 calls `Wait(P)` & P_1 calls `Wait(Q)` $\longrightarrow$ P = 0 & P = 0
2. P_0 calls `Wait(Q)` & P_1 calls `Wait(P)` $\longrightarrow$ P_0 & P_1 both become blocked as P = 0 & Q = 0 already
3. P is held by P_0 & Q is held by P_1 $\longrightarrow$ Signal of both processes cannot be reached to increment the semaphores (both processes reach deadlock)

1.6 Classical Synchronisation Problems

1.6.1 Producer-Consumer Problem

- **Overview:**
  - Processes share a bounded buffer of size K
  - Constraints: (1): Producers produce items to insert into buffer only when buffer is not full; (2): Consumers remove items from buffer only when buffer is not empty; (3): Producer should not produce when another producer is producing; (4): Consumer should not remove when another consumer is removing; (5): Producer & consumer should not produce & consume at same time
- **Naive Solution (Busy waiting):**

```
while (TRUE) {
   Produce Item;
   while(!canProduce);
   wait( mutex );
   if (count < K) {
       buffer[in] = item;
       in = (in+1) % K;
       count++;
       canConsume = TRUE;
   } else
       canProduce = FALSE;
   signal( mutex );
}
```
Producer Process

Critical Section

```
while (TRUE) {
   while (!canConsume);
   wait( mutex );
   if (count > 0) {
       item = buffer[out];
       out = (out+1) % K;
       count--;
       canProduce = TRUE;
   } else
       canConsume = FALSE;
   signal( mutex );
   Consume Item;
}
```
Consumer Process

  - Initially, `canProduce=`TRUE, `canConsume=`FALSE, `mutex=1`
  - Correctly solves problem, but busy waiting is still used in while loop
- **3 Semaphores Solution (Blocking version):**

```
while (TRUE) {
      Produce Item;

      wait( notFull );
      wait( mutex );
      buffer[in] = item;
      in = (in+1) % K;
      count++;
      signal( mutex );
      signal( notEmpty );
}
```
Producer Process

```
while (TRUE) {

      wait( notEmpty );
      wait( mutex );
      item = buffer[out];
      out = (out+1) % K;
      count--;
      signal( mutex );
      signal( notFull );

      Consume Item;
}
```
Consumer Process

  - `mutex`: binary semaphore that is used to acquire & release the lock (ensures only 1 producer/consumer in CS); `notFull`: general semaphore whose initial value == K (represents no. of empty slots); `notEmpty`: general semaphore whose initial value == 0 (represents no. of occupied slots)
  - Producer perspective: Producer calling `Wait(notFull)` sees notFull $>$0, indicating buffer still has space $\longrightarrow$ calls `Wait(mutex)` to acquire lock, inserts item, calls `Signal(mutex)` to release lock $\longrightarrow$ Producer calls `Signal(notEmpty)` to $\uparrow$ value of `notEmpty`, indicating 1 item inserted (If `notFull` == 0, buffer is full & producer is blocked until a consumer removes an item & calls `Signal(notFull)`)
  - Consumer perspective: Consumer calling `Wait(notEmpty)` sees notEmpty $>$0, indicating buffer still has items $\longrightarrow$ calls `Wait(mutex)` to acquire lock, removes item, calls `Signal(mutex)` to release lock $\longrightarrow$ Consumer calls `Signal(notFull)` to $\uparrow$ value of `notFull`, indicating 1 item removed (if `notEmpty` == 0, buffer is empty & consumer is blocked until a producer inserts an item & calls `Signal(notEmpty)`)

1.6.2 Reader-Writer Problem

- **Overview:**
  - Processes share a data structure where a Reader retrieves information & a Writer modifies information
  - Constraints: (1): Only 1 writer can access data structure (no other writer or reader should access at same time); (2): Reader can access data structure at same time as other Readers
- **Semaphore Solution:**

```
while (TRUE) {

    wait( roomEmpty );

    Modifies data

    signal( roomEmpty );
}
```
Writer Process

```
while (TRUE) {

    wait( mutex );
    nReader++;
    if (nReader == 1)
        wait( roomEmpty );
    signal( mutex );

    Reads data

    wait( mutex );
    nReader--;
    if (nReader == 0)
        signal( roomEmpty );
    signal( mutex );

}
```
Reader Process

  - `nReader`: Integer variable initialised to 0 (keeps track of current no. of readers), `roomEmpty`: initialised to 1 (keeps track of presence of readers & writers currently accessing data structure), `mutex`: initialised to 1, ensure mutual exclusion when `nReader` is updated (shared variable which can be updated by multiple readers)
  - Writer perspective: Writer calls `Wait(roomEmpty)`, indicating that no other reader/writer can access data structure $\longrightarrow$ Enters CS, modifies data $\longrightarrow$ calls `Signal(roomEmpty)`, other readers/writers can now access
  - Reader perspective: Reader tries to access data structure & increments `nReader` $\longrightarrow$ If `nReader` == 1, it means there is $\geq$ 1 reader present $\longrightarrow$ Reader calls `Wait(roomEmpty` which prevents writer from accessing data structure $\longrightarrow$ After reader is done, `nReader` is decremented $\longrightarrow$ If `nReader` == 0, there are no more readers present, can call `Signal(roomEmpty)`, allowing writer access
  - Reader is not bounded roomEmpty semaphores as multiple readers can access data structure at same time
  - Potential issue: If reader arrival rate $>$reader depart rate $\longrightarrow$ `nReader` may not be able to decrement to 0 & just keeps increasing $\longrightarrow$ writers will forever be blocked since `Signal(roomEmpty)` will not be called

1.6.3 Dining Philosopher Problem

- **Overview:** 5 philosophers are seated around a table, 5 single chopstick placed between each pair of philosophers $\longrightarrow$ When any philosopher wants to eat, he has to acquire both left & right chopsticks (Have to devise deadlock-free & starve-free way for philosopher to eat)
- **Naive method:**
  `Think(); takeChpStk(LEFT); takeChpStk(RIGHT); Eat(); putChpStk(LEFT); putChpStk(RIGHT);`
  All pick up left chpstk together $\longrightarrow$ each philosopher cannot pick up right chpstk(Deadlock occurs & cannot eat); Can make philosopher put down left chpstk if right chpstk cannot be acquired $\longrightarrow$ Left chpstk will be taken up & down repeatedly (livelock occurs)
- **Naive method with Mutex:** Surround above code with `Wait(mutex)` & `Signal(mutex)` $\longrightarrow$ but only 1 philosopher can take, eat, put at a time (inefficient)
- **Tanenbaum Solution:**

```
#define N 5
#define LEFT ((i+N-1) % N)
#define RIGHT ((i+1) % N)

#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[N];
Semaphore mutex = 1;
Semaphore s[N];

void philosopher( int i ){
    while (TRUE){
       Think( );
       takeChpStcks( i );
       Eat( );
       putChpStcks( i );
    }
}
```

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

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

void putChpStcks(i) { wait(mutex); state[i] = THINKING; safeToEat(LEFT); safeToEat(RIGHT); signal(mutex); }

  - `mutex`: binary semaphore that ensures only 1 philosopher can carry out 1 action; `s[N]`: array of binary semaphores with 1 semaphore per chpstck
  - Philosopher perspective: Become hungry & checks if it is safe to eat (if he is hungry & his neighbors are both not hungry $\longrightarrow$ calls signal on his own chpstck to show that he can acquire his chpstck as it is not used by his neighbors) $\longrightarrow$ Calls wait on his own chpstck so he can acquire his own chpstck if it is not taken by his neighbor OR become blocked if his own chpstck is taken by his neighbor $\longrightarrow$ proceeds to eat & puts down chpstick, change state to thinking & checks if it is safe to eat for his neighbors
- **Limited Eater Solution:** Just leave 1 empty seat (no deadlock can happen when 4 philosophers share 5 single chpstks as 1 philosopher can access 2 chpstks at 1 time)

1.7 POSIX Semaphores

- **Mutex:** `pthread_mutex` (Lock: `pthread_mutex_lock()`, Unlock: `pthread_mutex_unlock()`)
- **Conditional Variables:** `pthread_cond` (Wait: `pthread_cond_wait()`, Signal: `pthread_cond_signal()`, Broadcast: `pthread_cond_broadcast`)

2 Memory (Contiguous)

2.1 Basics of Memory

- **Memory Hardware**: Physical memory (RAM) can be treated as an array of bytes (each byte has unique physical address)
- **Memory Usage of Process:** Consists of Text (instructions), Data (global variables), Heap (dynamic allocation), Stack (function invocation)
- **Memory Management by OS:** OS allocates memory space to new processes, manage memory space for processes, protect memory space of processes from each other, provides memory-related system calls to processes, manage memory space for internal use

2.2 Memory Abstraction

- **Without Memory Abstraction:** Process directly uses physical address (No mapping needed, but 2 processes can occupy same physical memory if both processes assume memory starts at 0 $\longrightarrow$ hard to protect memory space)
- **Without Memory Abstraction (Address Relocation):** Recalculate memory references when process is loaded into memory by adding unique offset to all memory references in a process (but slow loading time, not easy to distinguish memory reference from normal integer constant)
- **Without Memory Abstraction (Base & Limit Registers):**
  - Base Register: Initialised to starting address of process memory space at loading time (All memory references are compiled as offset from base register)
  - Limit Register: Indicates range of memory space of current process
  - Cons: Every access: 1 addition (`Actual = Base + Addr`) & 1 comparison (`Actual < Limit`)
- **Logical Address:** How process views its memory space (Logical address != Physical address, mapping required)

2.3 Contiguous Memory Management

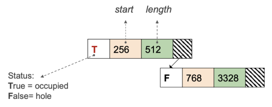
- **Assumptions:** (1): Each process occupies a contiguous memory region, (2): Physical memory is large enough to contain $\geq$ 1 processes with complete memory space
- **Memory Partition:** Contiguous memory region allocated to a single process (eg. RAM has 4 partition $\longrightarrow$ RAM can store 4 processes)   2 allocation schemes (Fixed size & Variable)

2.4 Fixed-size Partitioning

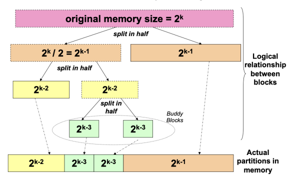
- OS maintains information about free & occupied partition via an ARRAY
- Memory is split into equal, fixed-size partitions (1 process occupies 1 partition)
- Pros: Easy to manage, fast to allocate (every free partition is same size, no need to choose)
- Cons: Partition size needs to be large enough to contain the largest process (Smaller process will waste space $\longrightarrow$ Internal Fragmentation)
- **Internal Fragmentation:** Process fails to occupy whole partition, left over space WITHIN partition is wasted

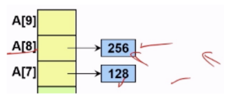
2.5 Dynamic Partitioning

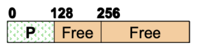
- OS maintains information about free & occupied partition via a LINKED LIST (list of partitions & holes with size values)



- Partition is created based on actual size of process
- Pros: Flexible, removes Internal Fragmentation issue
- Cons: Need to maintain more info in OS (linked list), takes more time to locate appropriate region, results in External Fragmentation
- **External Fragmentation:** Free memory space is created upon process creation/termination/swapping (wasted), but can be mitigated by merging holes by moving occupied partitions to create larger hole (can fit process)
- **Allocation Algorithms:** OS maintains a list of partitions & holes $\longrightarrow$ Need to locate partition of size N (size of process == N) $\longrightarrow$ Search for hole with size M $>$N $\longrightarrow$ Split hole into N & M-N (N is new partition, M-N is new hole)
  1. First-fit: Take 1st hole that is large enough (Fastest runtime since search stops once 1st hole found, but least efficient for memory usage as keeps assigning at beginning of list $\longrightarrow$ many holes at end of memory)
  2. Next-fit: Similar to First-fit, but search from last allocated block & wrap around circular linked list (ensures memory at the end is utilized too)
  3. Best-fit: Find smallest hole that is large enough (but slow runtime because need to search whole list for smallest hole)
  4. Worst-fit: Find largest hole (slow runtime because need to search whole list for largest hole, but more efficient memory-wise as less small holes formed after assigning a process $\longrightarrow$ can fit more processes in the future)
- **Buddy System Allocation:**
  - Overview: Free block is split into half repeatedly to meet request size (2 halves form buddy blocks), buddy blocks merge to form larger block when they are both free
  - Implementation Structure: An array `A[0...K]` is maintained where 2^K is largest block size that can be allocated, each `A[J]` is a linked list keeping track of free blocks of size 2^J, Each free block is indicated by starting address







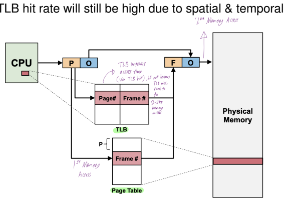
  - Allocation Algorithm:
    1. Find smallest S s.t. $2^S \geq N$
    2. Check if `A[S]` has free block
    3. Yes: Remove from list & return
    4. No: Find smallest R from S+1 to K s.t. `A[R]` has a free block $\longrightarrow$ For R-1 to S, keep splitting until there is a block in S $\longrightarrow$ Goto 2
  - De-allocation Algorithm:
    1. To free a block B, check in `A[S]` where 2^S == size of B
    2. If buddy C of B exists, remove B & C from list, merge B & C to get larger block B' $\longrightarrow$ Goto step 1, where B $\longleftarrow$ B'
    3. Else, insert B to list in `A[S]` (buddy of B is not free yet)
  - Locating Buddy: B & C are buddies of size 2^S if lowest S bits (0th - S-1th) of B & C are identical & Bit S is different (More sig. bits are identical too)
  - Analysis of Buddy System: (+): O(1) allocation & de-allocation time (since total # of memory blocks is fixed), (-): Has internal & external fragmentation

3 Memory (Disjoint)

ASSUMPTION REMOVED: Process fits in memory as contiguous piece $\longrightarrow$ Process can now fit into memory but split into disjoint chunks

3.1 Paging Scheme

- **Overview:**
  - Physical Frames: Physical Memory is split into regions of fixed size (Frames are small VS partitions are big to accommodate largest process memory space)
  - Logical Page: Logical Memory of process is split into regions of same size
  - Size of page == Size of frame (Pages of process are loaded into any available frame where logical memory is contiguous VS physical memory is disjoint)
- **Page Table:**
  - Maps Pages to Frames
  - Address Translation
    - Method 1: PA = FrameNo. * sizeof(frame) + offset
    - Method 2: PA = F * 2^n + O (given page/frame size of 2^n & m bits of LA $\longrightarrow$ P = most sig. m-n bits of LA, O = remaining n bits of LA $\longrightarrow$ use P to find F via page table)
    - Eg. Page size == 2^2 bytes & Page 00 maps to Frame 010, hence LA 00$\underline{01}$ maps to 010$\underline{01}$, where 01 is offset
- **Fragmentation:** External fragmentation is not possible (every free frame can be used), Internal fragmentation is insignificant (logical memory space may not be multiple of page size $\longrightarrow$ max 1 page per process not fully utilized)
- **Implementation of Paging Scheme:**
  - OS stores Page Table info in PCB (Page Table is per process, not globally shared across processes)
  - Memory context of process includes Page Table (or pointers to it since Page Tables are big)
  - Requires 2 memory accesses for every memory reference (1st access to read PTE to get frame number, 2nd access to access actual memory item)
- **Translation Look-Aside Buffer (TLB):** Aims to speed up address translation by acting as a cache for PTEs (TLB is a register in CPU, part of process' hardware context $\longrightarrow$ page table is huge, cannot store entire page table within TLB, but TLB hit rate will still be high due to spatial & temporal locality)



2nd Memory Access
CPU
P
O
F
O
TLB improves access time (via TLB hit), if not below; TLB miss, need to do 2-step memory access
Page#
Frame #
TLB
P
Frame #
1st Memory Access
Page Table
Physical Memory

- Address Translation with TLB: Use page number to search TLB $\longrightarrow$ TLB-Hit (Frame no. retrieved, generate PA) OR TLB-Miss (Memory access to access page table, retrieve frame no. to generate PA & update TLB)
- Avg. Memory Access Time (AMAT) = P(TLB_{hit}) * Latency(TLB_{hit}) + P(TLB_{miss}) * Latency(TLB_{miss})
  Latency(TLB_{hit}) = TLB Access + Actual Memory Access
  Latency(TLB_{miss}) = TLB Access + Page Table Access + Actual Memory Access
- Process Switching: During context switching, Page Table & TLB needs to be changed, but Frames remain same (TLB entries are flushed which causes many TLB misses to fill TLB when process resumes running)

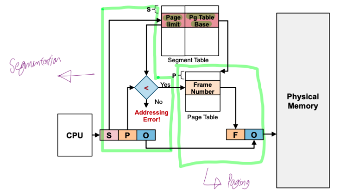
- **Protection Mechanism of Paging Scheme**
  1. Access-right bit: Each PTE has WRX bits (every memory access is checked against these bits in hardware; Child process has W = 0 initially for all pages after getting forked from parent which has W = 1 for some pages $\longrightarrow$ W only changes to 1 for child when writing is needed)
  2. Valid bit: Each PTE has Valid bit to indicate if page is valid to access by process (Some pages may be out of range for a process)
- **Page Sharing Mechanism of Paging Scheme:** Page table allows several processes to share same frame (Shared code page: Common code like standard lib, syscalls; Copy-on-write: Parent & child share a frame until 1 process tries to change value in it)

3.2 Segmentation Scheme

- **Overview:**
  - Logical memory space of a process == Collection of memory segments mapped into contiguous physical partitions of same size
  - Each memory segment has (1): Name, (2): Limit; All memory reference == SegmentName + offset
- **Segment Table:**
  - Contains 2 fields per entry: Base & Limit
  - Each segment is mapped to a contiguous physical memory region with a base address & limit
  - Segment Table is much smaller than Page Table (only a few segments, i.e. Data, Code, Stack, Heap)
  - Address Translation: Given LA = {SegID, Offset}, SegID used to look up {Base, Limit} of segment in Segment Table $\longrightarrow$ PA = Base + Offset (offset must be less than limit for valid access, or else seg fault error thrown)
- **Pros:** Each segment is independent contiguous memory space, segments can grow/shrink (by updating segment limit), be protected/shared independently
- **Cons:** External Fragmentation (variable size contiguous memory regions)

3.3 Segmentation with Paging Scheme

- Each segment is composed of several pages instead of a contiguous memory region $\longrightarrow$ Each segment has a Page table
- Segment can grow by allocating new page to add to its Page Table
- Segment Table's Base field is now base address of segment's Page Table instead of an address in physical memory



4 Memory (Virtual)

ASSUMPTION REMOVED: Physical memory is large enough to contain $\geq$ 1 process with complete memory space

4.1 Introduction

- **Overview:**
  - If Logical memory space of process $>$Physical memory, split logical address space into small chunks (Some chunks reside in physical memory, some stored in secondary storage)
  - Page Table is still used for Logical to Physical address translation (with addition of Resident Bit in PTE: Indicating whether page is resident in memory)
- **Page Fault:** Occurs when CPU tries to access non-memory resident page $\longrightarrow$ OS needs to bring non-memory resident page into physical memory (locate page in secondary storage, copy page into physical memory, update page table)
- **Benefits of Virtual Memory:** (1): Physical memory size no longer restrict size of logical memory address, (2): Page not needed can be on secondary storage, allowing efficient use of physical memory, (3): More processes can reside in memory ($\downarrow$ frames per process in physical memory)
- **Problems with Virtual Memory:**
  1. Thrashing: If memory access results in page fault most of the time, entire system slows down a lot ($\because$ secondary storage access time $>$physical memory access time) *Cost of loading page is amortised by temporal & spatial localities*
  2. Large no. of pages to allocate: Large startup cost when a new large process is launched *Solved by Demand Paging*
  3. Large Page Table: No. of pages can be very big *Solved by 2-level paging*
  4. Lack of frames to accommodate process: No. of pages can be very big, far exceed no. of frames in physical memory $\longrightarrow$ need to decide which page to replace *Solved by Page Replacement Algos*
  5. No. of frames to allocate to a process: Limited no. of physical memory frames $\longrightarrow$ need to decide how to distribute frames amongst processes *Solved by Frame Allocation Policies*

4.2 Demand Paging

- Aims to solve problem of large startup cost (Large no. of pages to allocate when new process is launched), reduce no. of frames per process
- **Mechanism:** Process starts with no memory resident page $\longrightarrow$ Only allocate a page when there is a page fault
- **Pros:** Fast startup time for new process, small memory footprint
- **Cons:** Process may appear sluggish at start due to multiple page faults, which may lead to thrashing

4.3 Page Table Structure

- **Overview:** Virtual memory results in huge logical memory space, huge no. of pages, huge page table $\longrightarrow$ Results in (1): High overhead (Every process has huge page table & page table is per process), (2): Page Table spanning several frames (less ideal as frames are scattered throughout physical memory space)
- **Direct Paging:** Keeps all entries in single table
  - e.g. Page size == 4KB, Virtual Addr is 64 bits long, Physical memory size == 16GB $\longrightarrow$ No. of virtual pages = $2^{64}/2^{12}$ = 2^{52}, No. of physical pages = $2^{34}/2^{12}$ = 2^{22} (A lot less frames than pages)
  - Given PTE size == 8B, Page table size = 2^{52}*8 = 2^{55}B per process (way higher than physical memory size, paging method is not feasible)
- **2-level Paging:** Aims to solve problem of Direct Paging by paging the page table (process may not use entire virtual memory space, full page table is wasteful)
  - Mechanism: VA is split into {Bits (M) for page directory #, Bits (P-M) for page #, Bits for offset#} $\longrightarrow$ Split Page Table into Page Tablets $\longrightarrow$ If original Page Table has 2^P entries & there are 2^M Page Tablets, each Page Tablet contains 2^{P-M} entries $\longrightarrow$ Single Page Directory keeps track of Page Tablets | Page directory can be more than 1 page long
  - Pros: (1): Less overhead (*Total overhead of page dir. + multiple page tablets $<$ total overhead of single large page table*), (2): Enables page table structures to grow beyond size of a frame (*Page dir. can point to page tablets residing in different disjoint frames*), (3): Can have empty entries in page dir. (*Corr. page tablets do not need to be allocated*)
  - Cons: 2 memory accesses needed to get frame no. (1st access for page dir., 2nd access for page tablet) $\longrightarrow$ TLB can eliminate page table accesses, but TLB misses will need to traverse more page tables
- **Hierarchical Page Table:** Radix tree structure, a table in each level of hierarchy is sized to fit in 1 frame, an invalid entry in any level means entire subtree does not exist

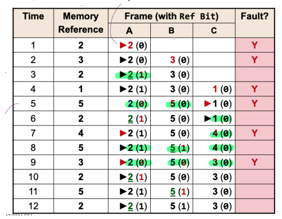
4.4 Inverted Page Table

- **Mechanism:** Keeps mapping of Frame # to {pid, Page #}, where pid == Process ID & Page # == Logical page no. in corr. process $\longrightarrow$ page # is not unique amongst processes, pid + page # can uniquely identify a memory page $\longrightarrow$ Entries are ordered by frame #, to look up Page X, need to search whole table
- **Pros:** 1 table for all processes, huge savings
- **Cons:** Slow translation
- In practice, Inverted Page Table is used as auxiliary structure (find out which processes & pages share frame X)

4.5 Page Replacement Algorithms

- **Overview:** No free frame during page fault $\longrightarrow$ need to evict a memory page (Clean Page: not modified, no need write back to storage; Dirty page: modified, need to write back to storage)
- Good Page Replacement Algo should minimize total no. of page faults

1. **Optimal Page Replacement (OPT)**
   - Replace page that will not be needed again for longest period of time (Kick out MAX next use time)
   - Guarantees min. no. of page faults
   - Problems: Not realizable as future knowledge of memory references is needed
2. **FIFO Page Replacement**
   - Replace page that has the oldest loading time (Kick out MIN loaded at time)
   - OS maintains queue of resident page numbers (1st page number gets kicked)
   - Problems: # of frames $\uparrow$ $\longrightarrow$ # of page faults $\uparrow$ (Belady's Anomaly) as FIFO does not exploit temporal locality, bad performance in practice
3. **Least Recently Used (LRU)**
   - Exploits temporal locality $\longrightarrow$ Replace page that has not been used in longest time (Kick out MIN last use time)
   - Aims to approx. OPT & does not suffer from Belady's Anomaly
   - Need to keep track of "Last Use Time"
     - Counter: Time counter that is incremented for every memory reference (Each PTE has a Last Use Time field), but need to search through all pages to find min. last use time, and overflow might occur as time-of-use keeps $\uparrow$
     - Stack: Maintain a stack of page no., if page X is referenced, remove X from stack & push X on top of stack, replace page at bottom of stack (no need search all entries), but this is not pure stack, hard to implement
4. **Second Chance (CLOCK)**
   - Modified FIFO give 2nd chance to pages that were accessed
   - Each PTE has a Reference Bit (1: Accessed since last reset, 0: Not accessed)



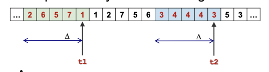
Time	Memory Reference	Frame (with `Ref Bit`)			Fault?
		A	B	C	
1	2	▶2 (0)			Y
2	3	▶2 (0)	3 (0)		Y
3	2	▶2 (1)	3 (0)		
4	1	▶2 (1)	3 (0)	1 (0)	Y
5	5	2 (0)	5 (0)	▶1 (0)	Y
6	2	2 (1)	5 (0)	▶1 (0)	
7	4	▶2 (1)	5 (0)	4 (0)	Y
8	5	▶2 (1)	5 (1)	4 (0)	
9	3	▶2 (0)	5 (0)	3 (0)	Y
10	2	▶2 (1)	5 (0)	3 (0)	
11	5	▶2 (1)	5 (1)	3 (0)	
12	2	▶2 (1)	5 (1)	3 (0)	

   - Algorithm:
     1. Oldest FIFO page is selected (Victim Page)
     2. If Ref Bit == 0, page is replaced
     3. If Ref Bit == 1, page is skipped, Ref Bit reset to 0, next FIFO page is selected, goto (2)
   - Degenerates into FIFO when Ref Bits of all pages == 1
   - Implementation: Circular list of all pages, pointer pointing to next potential victim page, finding a victim (advance until 1st page with '0' bit, clear any bits as pointer passes through)

4.6 Frame Allocation Policies

- **Overview:** Consider N frames & M processes competing for frames
  1. Equal Allocation: Each process gets N/M frames
  2. Proportional Allocation: Each process gets $size_p/size_{total}$ * N frames
- **Local VS Global Replacement**
  1. Local Replacement: Victim page is selected among pages of a process that causes page fault, thrashing is limited to 1 process (*Pros: # of frames allocated to process remains constant, stable performance btw. runs, Cons: Frames allocated may be insufficient*)
  2. Global Replacement: Victim page is selected among all frames (Process P can take frame from Process Q by evicting Q's frame) (*Pros: Process that needs more frames can get from those that need less, Cons: Badly behaved process can steal frames from other process, can cause other process to thrash too*)
- **Working Set:** Set of pages referenced by a process that is relatively constant in a period of time, occurs in stable region of Working Set Model
  - When function is executing, references are constant VS When function terminates, references change to another set
  - Aims to find right # of frames for a process

▪ Example memory reference strings



▪ Assume
  ❑ Δ = an interval of 5 memory references
▪ `W(t1,Δ)={1,2,5,6,7}` (5 frames needed)
▪ `W(t2,Δ)={3,4}` (2 frames needed)

5 File Systems Introduction

5.1 File System

- File system provides abstraction on top of physical media, high level resource management, protection btw. processes & users, sharing btw. processes & users
- **Memory VS File Management**
  Underlying Storage: RAM | Disk
  Access Speed: Constant | Variable disk I/O time
  Unit of Addressing: Physical memory addressing | Disk sector
  Usage: Address space for process, implicit when process runs | Non-volatile data, explicit access
  Organisation: Paging/Segmentation | ext(Linux), FAT (Windows), HFS(MacOS)

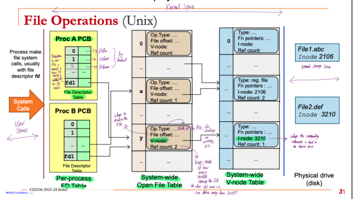
5.2 File

- **Basic Description:** File represents a logical unit of info created by process, contains Data & Metadata
- **File Metadata:** File Name (Human readable reference to file), Identifier (Unique ID for file used internally by FS), Type (Indicate different types of files such as ASCII, Binary files, directories | file types can be distinguished via file extension or magic number stored at file beginning), Size (current sizer of file in bytes, words or blocks), Protection (RWX permission bits for Owner, Group, Universe OR Access Control List), Time, date, owner info, Info for FS to determine how to access file
- **File Data:**
  - Structure: (1): Array of bytes *(Each byte has unique offset from file start)*, (2): Fixed Length Record *(Array of records, can jump to any record easily, where Offset of Nth record == Sizeof Record *(N-1))*, (3): Variable Length Record *(Flexible but harder to locate a record)*
  - Access Methods: (1): Sequential Access *(Data is read in order from beginning, cannot skip but can be rewound)*, (2): Random Access *(Data can be read in any order in bytes)*, (3): Direct Access *(Random access to any records directly, used for fixed length records, useful for large # of records*
  - Generic Operations: Create, Open, Read, Write, Repositioning *(move current position to new location)*, Truncate *(Removes data btw. specified position to EOF)*
- **System Calls:**
  1. `int open(char *path, int flags)`
  2. `int read(int fd, void *buf, int n)`
  3. `int write(int fd, void *buf, int n)`
  4. `off_t lseek(int fd, off_t offset, int whence)` (Offset(+) $\longrightarrow$ move forward, Offset(-) $\longrightarrow$ move backward | Whence: point of reference for interpreting offset, SEEKSET: absolute offset from start, SEEKCUR: relative offset from current position, SEEKEND: relative offset from end of file)
  5. `int close(int fd)` (fd is no longer used & kernel can remove associated data structures, fd can be reused later | Process termination automatically closes all open files)

  NOTE: Default File Descriptors: STDIN(0), STDOUT(1), STDERR(2)

5.3 File Information

- **Info for Opened Files:** (1): File Pointer (Keep track of current position within file), (2): File Descriptor (Unique ID of file), (3): Disk Location (Actual file location on disk), (4): Open Count (Tracks # of processes which has file opened)
- **Tables tracking file information (All in Kernel Space):**
  1. Per Process Open File Table: Found in PCB, keeps track of open files for 1 process, each entry points to SWOFT entry
  2. System wide Open File Table: Keeps track of all open files in system, each entry points to a V-node entry
  3. System wide V-node Table: Links with file on physical drive, contains info about file's physical location



- Process makes file syscalls with file descriptor
- Every time a new file is opened by a process, a new FD entry is added to process' PPOFT
- A file can be opened twice by 2 processes $\longrightarrow$ Need to have different entries for same file in SWOFT with different file offset (P0 may read/write to different portion of file from P1)
- Parent & Child processes use same SWOFT entry (only 1 offset needed)
- Open Count == 0 $\longrightarrow$ delete entry from table
- I-node: File data structure that stores info about file

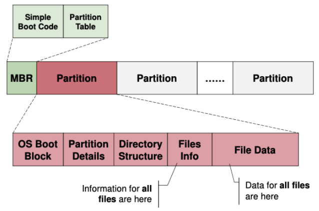
5.4 Directory

- Provides logical grouping of files, keeps track of files
- **Single Level:** Only can support unique names
- **Tree Structure:** Can have same file names as long as files are in different directories
- **Directed Acyclic Graph:**
  - A file can be shared (only 1 copy of actual content but appears in multiple directories, 2 file names referring to same file content)
  - Facilitated by Unix's Hard Link (Dir A is owner of File F, Dir B wants to share F $\longrightarrow$ A & B have separate pointers to F in disk)
  - Low overhead (Only pointers are added in directory), Deletion problems can be settled by Open Count (Delete file when 0), Cannot hard link to directories
- **General Graph:**
  - Cycles are allowed (Hard to traverse since need to prevent infinite looping, hard to determine when to remove file/dir)
  - Facilitated by Unix's Symbolic Link (special link file that contains path name of file)
  - If Symbolic Link is deleted $\longrightarrow$ Link file is deleted, but not file | If file is deleted $\longrightarrow$ File is deleted, but Link File remains as dangling link | Changing name of linked file will cause symbolic link to be dangling
  - High overhead (Link File requires a separate I-node & takes up disk space), Can symbolic link to directories

6 File Systems Implementation

6.1 Overview

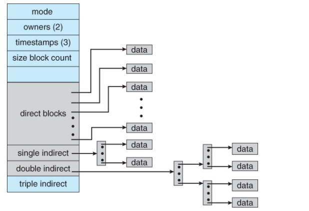
- **General Disk Structure:** 1D array of logical blocks (Logical Block: Small accessible unit, mapped into disk sectors) | 1 Directory is allocated a logical block | 1 file is split & allocated a few logical blocks
- **General Disk Organisation:**
  1. Master Boot Record: Located at sector 0 with Partition Table & Simple Boot Code (Activates partitions by checking if they have bootable device)
  2. Partition: Each partition contains an independent file system (eg. Windows in partition C:, Linux in partition D:)
  3. OS Boot Block: Code to load OS into RAM
  4. Partition Details: Total # of blocks, # & location of free/allocated blocks
  5. Directory Structure: Tracks files in a directory, map file name to file info
  6. Files Info: File Metadata
  7. File Data: File content



6.2 Implementing File (File Info + File Data)

Overview: A file is a collection of logical blocks $\longrightarrow$ Need to allocate file data to logical blocks via allocation schemes (Internal Fragmentation occurs when file size != multiple of logical blocks)

1. **Contiguous Allocation**
   - Allocates consecutive disk blocks to a file | exists a table in Files Info to keep track of {file name, start block #, length in terms of blocks} entries
   - Pros: Simple to track, fast access (Contiguous logical blocks are closely located on same/adjacent disk sectors, no need to move disk head a lot)
   - Cons: External Fragmentation (Holes created btw. logical blocks occupied by files), File size must be specified before (fit file in a appro. range of contiguous logical blocks)
2. **Linked List Allocation**
   - Linked list of disk blocks (Each disk block stores next disk block # via pointer & actual file data) | Last block in linked list has special value | exists a table in File Info to keep track of {File Name, Start Block #, End Block # } entries (End block makes appending operation faster coz no need traverse linked list)
   - Pros: Solves Fragmentation problem (Logical blocks need not be contiguous)
   - Cons: Random access in file is slow (blocks can be in different disk sectors), Additional overhead to store pointers (part of disk block used for pointer), Less reliable (pointers may be wrong)
3. **Linked List V2.0 FAT Allocation**
   - Move all block pointers into File Allocation Table (FAT) which is in memory
   - Exists a table to track {File Name, Start Block #} (No need store End Block # since FAT is more efficient) | FAT entry is of form {Current Block #, (FREE, Next Block #, EOF or BAD)}
   - FAT Version: FAT16 (2^{16} entries, entry size == 16 bits since address of each FAT entry is stored) | FAT32 (2^{32}, entry size == 32 bits)
   - Pros: Faster random access (Linked list traversal is in memory instead of hard disk)
   - Cons: Consume lots of memory (FAT tracks all disk blocks in partition)
4. **Indexed Allocation**
   - Each file has an index block (contains an array of disk block addresses, IndexBlock[N] == Nth block address, Each address is pointer to actual logical block)
   - Pros: Less memory overhead (only Index Block needs to be in memory), Fast direct access (Directly access via array indexing, no linked list traversals)
   - Cons: Limited max file size (max # of blocks == # of Index Block entries), Index Block overhead (block is used for Index Block instead of data storage)
5. **Indexed Allocation Variants**
   - Allow for larger files (via multiple index blocks)
   - Linked List Scheme: Keep linked list of Index Blocks (but expensive due to traversal cost)
   - Multi-level Index: Similar idea to multi-level paging
   - Combined Scheme: Use Direct Index (fast access to small files), Multi-level index (large files)
   - Eg. of Combined Scheme: UNIX I-Node



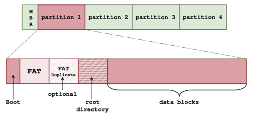
   - Every file/dir has a I-node data block associated to it (I-node consists of direct, indirect blocks)

6.3 Free Space Management (Partition Details)

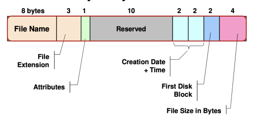
- **Overview:** Need to know which disk block is free to perform file allocation (Partition Details maintain free space information, remove free disk block from free space list for allocate, add free disk block to free space list for free)
- **Bitmap:** Each disk block is represented by 1 bit (Bit 0: occupied block, Bit 1: free block | provides good set of manipulation but need to keep in memory for efficiency reason)
- **Linked List:** Linked list of disk blocks (Each disk block contains a # of few disk block numbers OR a pointer to next free space disk block | Easy to locate free block, only 1st pointer needs to be in memory, but high overhead)

6.4 Implementing Directory (Directory Structure)

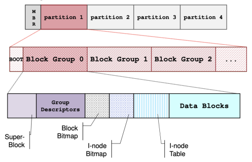
- **Overview:** Keeps track of files in directory & maps file name to file info | Sub-directory stored as file entry with special type in a directory
- **Linear List**: Directory consists of a list where each entry represents a file (stores file name & metadata & file info OR pointer to file info) $\longrightarrow$ but requires linear search (inefficient for large directories)
- **Hash Table:** Directory contains a hash table of size N where File Name is hashed into index K where 0 $\leq$ K $\leq$ N-1 (Fast lookup, but may have collisions)
- **File Info:** (1): Store all file info in a directory entry, (2): Store only file name & points to some data structure for other info
- **FAT16 Layout:**



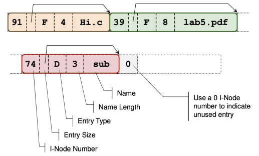
- **FAT16 Directory Entry:**



- **Ext2 Layout:**



- **Ext2 Directory Entry:**



6.5 File System in Action

File Creation (Create File F):
1. Use full path name to locate parent directory
2. Search for file name F to avoid duplicates
3. Use Free Space List to find free disk blocks
4. Add an entry to parent directory (with relevant file info, file name, disk block info)

File Opening (Process P opens File F):
1. Use full path name to locate File F
2. When F is located, F's file info is loaded into new entry E in system-wide table & V in I-node table if not already present
3. Create entry in P's table pointing to E & make E point to V
4. Return file descriptor for further read/write operation