Swapping:

So far, everything is reduced into one memory map. All programs irrespective of their sizes reside into the memory map. But, there are some problems in this structure,

1) If big programs are loaded into memory then it can lead to the lack of space in the memory map as we try to load a new program.
2) Sometimes, programs are idle while in IO wait and do nothing for long time and still keep the memory space with them.
3) Memory map is fragmented due to invariant size of the program in the memory.

Swapping is used to solve these issues by stashing out big and idle programs out of the memory map and take them to some disk space.

Simple Swapping

In the Scheduler Code,

```
Every once in a while
If a process has been in I/O wait for long time
Start swapping it.

Get Input
If destination is swapped
Start swapping it in.
```
**What triggers Swapping?**
- Once a while as mentioned in the scheduler code
- If while creating a process no memory available, then do swapping of idle process

---

**Pipes**

Definition: Pipe the output of one command to the input of other command

Example: `ls | grep <filename>`

Implementation of Pipe in MS-DOS:
- Take the output of one command and write it into the file
- Read the file so to have input for the next command

The above method is the simplest way to implement pipes but not the efficient. The buffer for this implementation is the preferred way, often used.

![Pipe Buffer Diagram](image)

**Write**
- If full
  - `Pipe.writer = self`
  - Wait
  - Append to buffer
- If `pipe.reader`
  - `Signal pipe.reader`

**Read**
- If Empty
  - `Pipe.reader = self`
  - Wait
  - Get Data
- If `pipe.writer`
  - `Signal pipe.writer`

*Note: This code for one character at a time, but a loop can be used to make it working for whole program*

When the buffer is empty, the process goes into IO wait state and when pipe reader is signaled in writer the process goes into **Runnable** state from **IO wait** state. So, it doesn’t require any device or terminal for pipes.

Example: `ls | grep <filename> | more`
Threads

Threads are light weight processes. Multiple threads can be run in a program. Usually every process has at least one thread running in it.

Why Use Threads?

- **Parallelism** → Multiple threads can execute single code or different code in the program so to improve performance of the processor. It leads to a truly concurrent execution of a program.

- **Robustness** → if one thread crashes then other thread can take over the operation so to avoid a process crash.

- **Light Context Switch** → Context switch in threads within a process is much faster and less costly than that in processes.

- **Common Data** → Multiple threads, on the other hand, typically share the state information of a process, and share memory and other resources directly.

- **Better Resource Allocation**

- **Asynchronous Operations**
  - Multiple Logically Separated processes
  - Single Program reads from different users
  - Web servers

- **Sharing Data Structures**

Critical Section Problem

The above problem could lead to the increment of X twice when we want it to be incremented by one. There is no exclusive lock in those two threads. The problem of critical section can be described as below:

- Multiple threads are competing to used some shared code/data
- Each thread has a code segment which is called critical section and this section contains the shared data accessed.
- Problem is to make sure when one thread is executing in its critical section then no other thread is allowed to execute in its critical section

Solutions to Critical Section Problem:

- Disable Interrupts (it works in uni-processors only)
- Atomic Instructions e.g. ATOMIC_INCR, CMPXCHG etc
- MUTEX
- Semaphores
- Monitors
- Events
- Spin Lock

**Mutex: (Mutual Exclusion)**

Mutex is good for protecting shared data; it basically takes the critical section problem and implements it as it is. When one thread is executing into its critical section then it doesn’t allow other threads to execute in their critical sections.

```plaintext
Index_lock (my mutex)  
................ // Critical Section  
Mutex (unlock)
```

**Semaphores:**

Semaphore is a protecting variable for restricting access to shared resources or shared memory.

```

P (Semaphore s) // Acquire Resource
{  
    wait until s > 0, then s := s-1;
}

V (Semaphore s) // Release Resource
{  
    s := s+1;
}

Init (Semaphore s, Integer v)
{  
    s := v;
}
```

**Monitor**

Only one thread can be inside the critical section once.

- Condition variables

```
PThread_mutex_lock (m)  
While condition is not true
    Sleep_on (m, condvar)
mutex_unlock (m)

mutex_lock (m)  
modify condition
signal_condvar(condvar)
mutex_unlock (m)
```
Example: If thread_can_run = 5,

```
pthread_mutex_lock (m)
While threads_can_run < 1
        sleep_on (m, condvar)
        thread_can_run --;
mutex_unlock (m)
```

note that this is an implementation of a counting semaphore

**Spinlock**

Thread waits in a loop (or spins) repeatedly checking until the lock becomes available. Spinlock protects small data structures but waiting for long time is a waste.

--------------------------------------------------------------------
End of Scribe Notes for this part
- Prepared By Sudershan Bhandari