Made by Ivan Koh and Li Mingyang

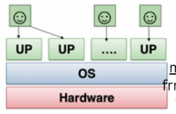
Introduction to OS

OS: Program that acts as an intermediary between a computer user and computer hardware

OS evolves with:
1. Computer hardware
2. User application
3. Usage pattern

First OS:
+ Minimal Overhead as programs interact directly with hardware
- Not portable
- Inefficient use of computer

Batch OS: Execute job sequentially, inefficient use of CPU



Layered Systems: Genrlztn of monolithic system. Organise compnts -> layer hierarchy

Client-srvr model: variatn of microkernel. client proc req sv frm srvr proc. srvr proc built on top of microkernel. both proc can be on separate machine

Time Sharing OS:
+ Multiple users can interact with machine using terminals
+ User job scheduling (with illusion of concurrency)
+ Shares CPU time, memory, storage
+ Each program executes as if it has all resources to itself

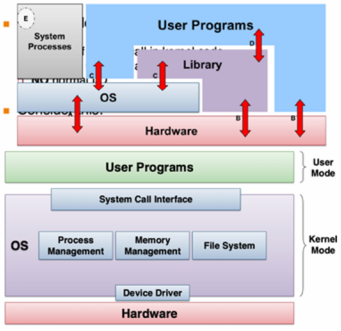
similar to Unix servers tdy

Motivations of OS: allow simult. exec of program
1. Manage resources and coordination (process synchronization, resource sharing), resolve conflict
2. Simplify programming (abstract hardware and low level details) perform tasks thru OS
3. Provides security and protection
4. Prevent error and improper use
5. Efficiency, portability

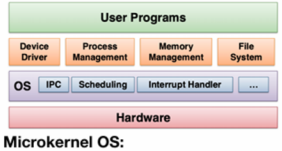
Implementing: w HLLs (C/C++), v hardware archi-dependent.

OS Structure:
- OS is also known as the **kernel**
  - A program with some special features
    - Deals with hardware issues
    - Provides **system call interface**
    - Special code for interrupt handlers, device drivers

Challenges:
- "No one else" to rely on for nice services
- Debugging is hard
- Complexity
- Enormous Codebase



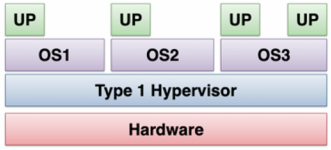
Monolithic OS:
- One big special program (doesn't mean it is compiled all together!)
- Approach by Unix variants and Windows NT/XP
- IO/ CPU/ File system etc done in Kernel mode
- More efficient as do not need to switch between User and Kernel mode => Less clock cycles
- Written by many people => could be buggy
- Change require compiling of entire kernel => Less extensible



Microkernel OS:
- Minimal kernel with only basic services (IPC/ scheduling etc.)
- Less efficient due to IPC overhead and more context switches
- More secure => buggy code in user mode services do not crash the OS
- Approach by MacOS

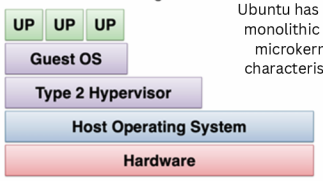
Why VM? -> OS assumes total control of hardware. What if want to run many OS at once? -> OS hard to debug/monitor so use VM

Running OSes: Virtual Machines aka Hypervisor
1. Software emulation of hardware
2. Used to run multiple OS on the same hardware
3. Each OS has the illusion of maintaining control over the machine
4. Offers protection for the machine if there are bugs in testing OS



Type 1 Hypervisor:
1. Runs directly on hardware to give support to multiple OS
2. Individual VMs to guest OSes

Ubuntu has both monolithic and microkernel characteristics



Type 2 Hypervisor:
1. Remapping cores from Guest OS to Host OS
2. Running Guest OS on Host OS
3. Goes through Host OS first => inherently slow

Process Abstraction

Memory space consists of:
1. Text Memory: For instructions
2. Data Memory: For global variables
3. Heap Memory: For dynamic allocation (e.g. using malloc)
4. Stack Memory: For function invocations (e.g. local variables)

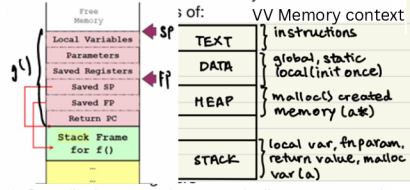
Cores <> Processes: m CPUs has at most m processes

Process: Dynamic Abstraction for executing program, containing information to describe a running program

Information: Memory Context, Hardware Context, OS Context

For ptrs: Ptr itself is in stack, but mem location ptr pts to is in heap.

Stack Memory:
1. Memory region to store information on function invocations
2. This information is described by a **stack frame**



4. Compiler inserts code automatically to create stack frame when a function is called

Stack Pointer:
1. Top of stack region (first unused location) is indicated by a Stack Pointer
2. Stack frame is added **on top** when function invoked
3. Stack frame removed **from top** when function ends

Frame Pointer-> not all OS have, platform dependent
1. To facilitate access of various stack frame items as stack pointer can shift
2. Points to a **fixed location** in stack frame (usually the start of the stack frame)
3. Other items are accessed as displacement from the frame pointer

Stack Frame Setup & Teardown

Caller:	Push $fp and $sp to stack
	Copy $sp to $fp
	Reserve space on stack for parameters by moving $sp
	Write parameters to stack using offsets of $fp
	JAL to callee
Callee:	Push $ra to stack
	Push current register values used in function
	Use $fp to access parameters
	Compute results
	Write result to stack (where $fp is)
	Restore registers saved
	Get $ra from stack
	Return to caller using JR $ra
Caller:	Get result from stack
	Restore $sp and $fp

Register spilling: move register value to memory temporarily when no registers available

Note: register value moved into saved registers section in stack

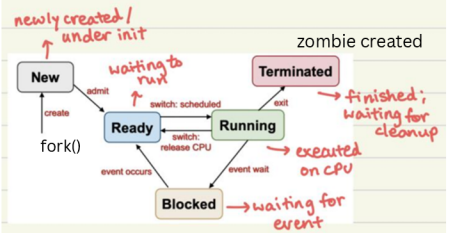
Heap Memory:
1. Allocated only at runtime => Cannot place in data region
2. No definite deallocation timing => Cannot place in Stack region
3. Heap memory are allocated/deallocated in such a way to create "holes" in memory
   1. Free memory block squeezed between occupied memory block

Process Identification:
1. Used to identify processes (using PID)
2. Unique among processes
3. PID 1 reserved to a process called init

Process State:
1. Used to identify processes (using PID)
2. Unique among processes
3. PID 1 reserved to a process called init

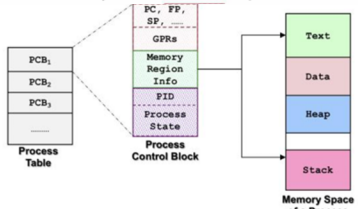
Process Model:
1. New: Looks at .exe header and tries to figure out how much memory (RAM) is required, new process created
2. Ready: Process waiting to run
3. Running: Process being executed on CPU
4. Blocked: Process waiting for event (e.g. I/O), cannot execute until event is available
5. Terminated: Process finish execution, all memory resources released

Process scheduling



Zombie: child process terminates but parent does not call wait()
Orphan: parent process terminates before child process
- init process becomes pseudo-parent (handles termination)

Process Control Block (PCB)
- Entire execution context for a process
- Maintained by kernel; stored under **process table**



System calls -> API to call svcs in kernel, diff OS hv diff APIs
- Must change from user mode -> kernel mode

Function Wrapper	Function Adapter
Syscall with library version with same name and parameters	User friendly syscall with less params/more flexible param

Mechanism:
1. User program invokes library call
2. Library call places system call number in a designated location like a register
3. Library call executes special instruction to switch from user to kernel mode (TRAP), saving the CPU state
4. Appropriate system call handler is determined by system call number (index), handled by dispatcher
5. System call handler executed
6. System call handler ends, restoring CPU state and returning to library call, switch back to user mode
7. Library call return to user program

Exceptions
1. Occurs when executing a machine level instr (e.g. add/ sub)
2. Exception is synchronous, and occur due to **program errors**
3. When it occurs, exception handler executed automatically (like a forced functional call)

Interrupts
1. Caused by **external events**, usually **hardware related**
2. E.g. Timer, Keyboard Pressed (Ctrl + C)
3. Interrupt is asynchronous, and can occur independent of program execution (since hardware independent of software
4. When it occurs, program execution is suspended, interrupt handler executed automatically
5. All hardware has its own interrupt handler, keyboard, timer has its own interrupt handler

```
void handler()
{
  1. Save Register/CPU state
  2. Perform the handler routine
  3. Restore Register/CPU
  4. Return from interrupt
}
```

1. If handled successfully, handler fixes the issue, program execution resumes and it may behave as if nothing happened
2. If it is unrecoverable, e.g. seg fault, process is killed

Process Abstraction in Unix

1. Contains PID, Process State (Running, Stopped, Zombie etc.), PPID, Cumulative CPU time etc.

fork() ts duplicates EVERYTHING, including recursion back to main

```
int fork();
```

- Returns PID of child process (in parent) OR 0 (in child)
- Creates new process, duplicates the current executable machine; executes the remaining code below the fork()
- Memory in child is copied from parent (copy-on-write)

Implementation:
1. Create address space of child process (virtual address space)
2. Allocate p' = new PID
3. Create kernel process data structures (process table entry)
4. Copy kernel environment of parent process (priority)
5. Initialize child process context: PID = p', PPID = parent PID, 0 CPU time
6. Copy memory regions from parent (expensive, optimised with copy-on-write)
7. Acquires shared resources (open files, pwd)
8. Initialize hardware context for child process (copy registers from parent process)
9. Ready to run (add to scheduler queue)

Copy on Write: optimization for memory copying operation where memory locations that are written to are duplicated; noop if reading only . Some scenarios whr partial duplication is preferred.

int execl() Command
1. To replace current executing process image with a new one
2. PID and other information still the same
3. E.g. Code in child replaced with code that you execl()
4. Format: int execl(const char *path, const char *arg0,..., NULL
   1. Path: Location of executable
   2. Arg0…argN: command line arguments
   3. NULL: indicate end of argument list
5. e.g. execl("/bin/ls", "ls", "-l", NULL); === ls -l
6. Using fork() + execl(), we can **get a new process for running a new program**

exit() -> takes in a status in arg, that is returned to parent process.

```
void exit(int status);
```

- Ends execution of process, does not return control to caller
- Status 0 success, !0 error
- Most system resources released like file descriptors
  - Left with PID & status, PCB
- Implicitly done when returning from the main() function

wait()

```
int wait(int *status);
```

- Parent waits for immediate child process (only) to terminate
- Returns PID of terminated child process
- Status is passed by address (wait(&status)); NULL if unused
- Blocking behavior on parent until one child terminates
- Clears remainder child sys resources, kills zombie processes
- execl() does not affect wait()
- If not children, terminate

remainder resources eg PID, status, stuff not removed on exit()

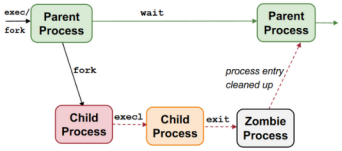
Parent alr retrieved child's exit status, so dont need to keep anymore

waitpid(): wait for specific child, waitid(): wait for any child to change status

Zombie Process
1. Parent process terminates before child process:
   1. init process becomes "pseudo" parent of child process
   2. Child termination sends signal to init, which utilizes wait() to cleanup (so can still cleanup eventually!)
2. Child process terminates before parent but parent did not call wait:
   1. Child process becomes a zombie process
   2. Can fill up process table and may need reboot to clear table!

Master Process
1. init process, created in kernel at boot up time (has PID = 1)
2. Watches for other processes and respawns where needed (especially when important processes die)
3. fork() creates process tree, with init as the root process

Implementation Issues of fork()
1. Need to copy memory regions from parent (Very expensive!)



time slicing: interleaving instn for multiple processes. Context switching btw processes incurs overhead

Process Scheduling

Concurrent Processes
1. Multiple processes that progress in execution at the same time
2. Could be virtual parallelism (One CPU doing many things tgt)
3. Could be physical parallelism (Multiple CPUs doing things)

Terminology

Scheduler: Part of OS that makes scheduling decision
Scheduling algorithm: Algorithm used by scheduler
Timer Interrupt: Goes off periodically (based on hardware clock); OS ensures that it cannot be intercepted by other program
Interval of Timer Interrupt: Scheduler triggered every timer interrupt; typically 1ms to 10ms
Time Quantum: Execution Duration given to a process; Could be constant or variable; Must be **multiples of ITI;** 5 to 100ms

Scheduling
1. Multiple ways to allocate processes (influenced by **process environment**, scheduling algorithms)
2. Each process has different CPU time requirement (process behavior)

Process Environments

Batch processing: No user interaction, no need to be interactive	**Turnaround time:** finish – arrival **Waiting time:** time spent in queue **Throughput:** number of tasks finished per unit time **CPU utilization:** % time where CPU is not idling
Interactive: Active users interacting with system, must be responsive	**Response time:** first CPU use – arrival time **Predictability:** variation in response time
Real-time processing: Deadline to meet, usually periodic process	

Scheduling Algo Criteria: Fairness: all process gets fair share of CPU, no starvation. Utilization: all parts of system should be used. (CPU, I/O)

Types of Scheduling
1. Non-preemptive (cooperative): process keeps running until blocked or gives up CPU voluntarily
2. Preemptive: process given fixed time quantum to run
   - Can be blocked or give up early
   - Forced to suspend once time quantum up
   - Ensures good response time as scheduler runs periodically
   - Periodic timer interrupt that cannot be intercepted; invokes scheduler

Step by Step Scheduling Process
1. Scheduler triggered (OS takes over)
2. If context switch needed, context current running process is saved and placed in blocked/ready queue
3. Pick a suitable process to run base on scheduling algorithm
4. Setup context for P, and let process P run

Scheduling Algorithms

turnaround time: finish - come in

FCFS

Batch processing, non-preemptive

Tasks stored in queue based on arrival time & tasks run till done or blocked

Convoy effect: 1. ReadyQ: task A(cpu bound) followed by a lot of I/O bound tasks X 2. A runs first, and everyone wait(I/O idle) 3. A finished CPU and send to I/O queue. The rests of the tasks X finishes quickly now cpu is idle; poor utilisation

Blocked tasks are removed from queue and when I/O completes, added back to queue

Guarantees no starvation as every task runs

Average waiting time can be improved and convoy effect present where long CPU bound task blocks I/O and vice versa

Shortest average response time if jobs arrive in order of increasing lengths or all jobs have same completion time

Batch processing, non-preemptive

Select task with smallest total CPU time (needs this info in advance/guess) **SJF - s job first**

Prediction algorithm from previous CPU-bound phases using exponential average

$$Predicted_{n+1} = \alpha Actual_n + (1-\alpha) Predicted_n$$

α: weight placed on recent or past event

Minimizes average waiting time given a fixed set of tasks actualn: the most recent cpu time consumed. Predictedn = the past history of CPU time

Starvation is possible due to bias towards short jobs

Batch processing, preemptive

Variation of SJF using remaining time **SRT - s rem aining time**

New jobs with shorter remaining time pre-empts currently running jobs; good for short jobs with late arrival

Interactive, preemptive **RR - Round Robin**

Like FCFS but preemptive so when time quantum lapses, task forced to give up CPU and placed to end of queue (after I/O)

Response time (guaranteed) to be upper bounded by (n - 1)q with n tasks and time quantum q

Larger time quantum yields better CPU utilization but longer waiting time, shorter time quantum yields larger context switching overhead but shorter waiting time **Thread growth**:user library->OS feature->hardware feature (simultaneous multi threading (**SMT**))

Behaves like FCFS if job lengths are shorter than time quantum

SMT ONLY : 1 CPU can execute 2 threads

Performs worse than FCFS if job lengths are all the same and greater than time quantum (higher average turnaround time) or when there are many jobs and job lengths exceed time quantum (reduced throughput due to increased overhead)

Priority Scheduling

Interactive, preemptive/non-preemptive

Assigns priority to processes and run highest priority first

5. **Hard to control the exact CPU time** given to a process using priority

Preemptive: Higher priority process can preempt running process with lower priority

Non-preemptive: Late coming waits for next round of scheduling Higher priority job will NOT ALWAYS preempt lower priority job due to priority inversion

Low priority process can starve if high priority keeps hogging CPU (worse for preemptive); resolved by decreasing priority or current process after each time quantum or giving current process a time quantum

Priority inversion: lower priority preempts higher priority because resource is locked so higher priority is blocked

Interactive, preemptive **Multi-lvl Feedback Q (MLFQ)**

Multiple queues with different priority levels, minimizes response time for I/O bound processes and turnaround time for CPU bound processes

Highest priority queue must be empty before next queue is used

Allow scheduling without perfect knowledge as it learns the process behavior

Rules:

1. Priority(A) > Priority(B) -> A runs
2. Priority(A) == Priority(B) -> A and B runs in RR
3. New job -> highest priority
4. Job fully used time quantum -> reduce priority
5. Job gives up before time quantum -> retain priority

Change of heart: process with lengthy CPU phase right before I/O phase sinks to the lowest priority and gets starved; periodically move all tasks to highest priority to fix

Gaming the system: process repeatedly gives up CPU before time quantum lapses remains in high priority and starves other processes; accumulate total CPU use time across all quantums

Interactive, preemptive **Lottery Scheduling**

Give out "lottery tickets" to processes for system resources, randomly choose lottery ticket, winner gains resource; lottery tickets can be distributed t children

Process holding X% of tickets has X% chance to win and use X% of resource

Properties

Responsive: A new process can participate in the next lottery

Good level of control: Process can be given Y lottery tickets, which it can then distribute to its child process

(E.g. even if a process has 100 children, that process will still consume fix amount of CPU time.)

Each rsc can hv own set of tickets -> diff proportion of usage per rsc per task

Process Alternative - Threads

Motivation

1. Expensive process creation, context switching
2. **No easy way to communicate** between processes due to independent memory space

Basic Idea

1. Single process can have multiple threads
2. Can share **memory context** and **OS context** between threads
3. Only need to perform **hardware context switch** [Registers and Stack (actually just stack pointer and frame pointer)] for threads

Unique Information for each thread

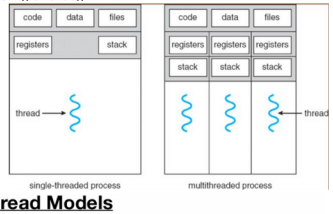
1. Identification (usually thread id)
2. Registers (General purpose and special)
3. "Stack" NOTE: threads in same process share all memory EXCEPT stack memory_

Benefits

1. Economy: Multiple threads in same process **requires much less resources** to manage compared to multiple processes
2. **Resource sharing**: No need for additional mechanism for passing information around
3. **Responsiveness:** Multi-threaded programs appear more responsive
4. **Scalability:** Multi-threaded program can use multiple CPUs (OS can schedule one thread on one CPU)

Problems:

1. System call concurrency: ensure that parallel system calls have no race condition
2. Process behavior: what happens when a single thread calls exit() or exec()



Thread Models

User Thread

- Thread implemented as a **user library**
- Runtime system (in process) handles thread-related operation
- Kernel is **not aware of threads** in the process

Advantages:

- Can have multi-threaded programs on ANY OS
- Thread operations are just library calls
- More configurable and flexible
- **Do not require syscall** as everything is running within the process => much more efficient

Disadvantages:

- OS not aware of threads, scheduling performed at process level
- 1 thread blocked => Process blocked => All threads blocked
- **Cannot exploit multiple CPUs**

Kernel Thread

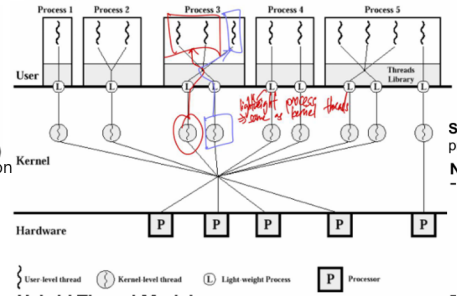
- Thread implemented in the OS, thread operations handled as syscalls
- Kernel can use threads for its own execution

Advantages:

- Thread-level scheduling possible: Kernel can schedule by threads, instead of by process; More than 1 thread in same process can run simultaneously on multiple CPUs

Disadvantages:

- Thread operations is now syscall => slower and more resource intensive
- Less flexible (if implemented with many features, overkill for simple program; if implemented with few features, not flexible enough for some programs)



Hybrid Thread Model

- Have both Kernel and User threads
- OS schedule on kernel threads only
- **User thread** bins to **kernel thread**

Advantages:

- Offer great flexibility and can limit concurrency of any process

Simultaneous Multi-threading

- Supply multiple sets of registers to allow threads to run natively and in parallel on the same core

Threads on Modern Processor

- Hardware support on modern processors:
- Supply multiple sets of registers (GPRs and special registers) to allow threads to run natively and in parallel on the same core

POSIX Threads: pthread

- pthread can be implemented as user/kernel mode
- pthread_t: Data type to represent **thread id (TID)**

pthread_attr: data type to represent attributes of thread

pthread_create:

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

Return 0 if successful, !0 if errors

tidCreated: thread id for the created thread
threadAttributes: control the behavior of the new thread
startRoutine: function pointer to the function to be executed by thread
argForStartRoutine: arguments for the startRoutine function

pthread_exit: pthread terminates automatically at the end of the startRoutine with return value defined by return statement

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

exitValue: value to be returned to whoever synchronize with this thread if pthread_exit() not used, thread auto terminates but does not return a value

pthread_join: wait for the termination of another pthread

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

Return 0 if successful, !0 if errors

threadID: TID of the pthread to wait for
status: exit value returned by the target pthread

Inter-Process Communication

- Hard for cooperating processes to share information as memory space independent

Shared Memory

- Process A **creates** a shared memory region M.
- Process B **attaches** M to its own memory space. (OS involved till here) Now, A and B can communicate using the shared memory region M.
- Any writes can be seen by **all other parties**

Advantages

- Efficient (Only creating + attaching requires OS)
- Ease of use (Behaves similar to normal memory, can read/write data easily)

Disadvantages

- Synchronisation: Need to sync access of shared resources
- Harder Implementation

POSIX Shared Memory in Unix:

- **shmget:** Creates shared memory region - Returns segment ID of shared memory segment created if successful, else -1. Generally done by master programs
- **shmat**: Attaches shared memory segment to addr space of calling process so memory contents can be accessed. Returns address of attached segment else -1.
- **shmctl**: Used to perform one of several control operations on the shared memory segments. Returns 0 if successful, else -1. If IPC_RMID indicated, it removes shm segment and its ID from system after all users have detached

Message Passing

Process A prepares message M and send to Process B. M is **stored in kernel memory space**. Process B receives message M. Both send and receive are syscalls and have to **go through OS.**

Advantages:

- Portable: easily implemented across multiple OSes
- Easier synchronization: Implicit synchronization defined by synchronization behaviors below

Disadvantages:

- Inefficient: Every send/receive requires OS
- Hard to use: Message limited in size and type

Synchronization behaviors for send() and receive(): blocking primitives (synchronous) or non-blocking primitives (asynchronous)

Naming Scheme

- **Direct Communication:** Sender/Receiver explicitly names the other party
  - Requires **1 link per pair** of communicating processes
  - Need to know the identity of other party
  - e.g. send(B, Msg), receive(A, Msg)
- **Indirect Communication:** Message sent/received from message storage (known as mailbox MB)
  - One mailbox can be shared among number of processes
  - e.g. Send(MB, Msg), Receive(MB, Msg)

Unix Pipes (IPC)

- Share input/output between processes (producer/consumer)

Process communication channels: stdin, stderr, stdout

Piping (A | B): links the input/output channels of one process to another V semantic V

Pipe in C: circular bounded byte buffer with implicit synchronization (writer waits when buffer is full, reader waits when buffer is empty)

Unidirectional (half-duplex): one write end, one read end

Bidirectional (full-duplex): any end for read/write

Possible to redirect the input/output from one program to another

System calls:

int pipe(int fd[])	read is blocking. (will block until data comes in pipe)
close(int) -> side to close	
write(int, data, length) -> side to write to	
read(int, buffer, length) -> read from	
0 for success, !0 for errors	close(fd[0], sends SIGPIPE signal to other side
fd[0] is reading, fd[1] is writing	close(fd[1]) sends eof to other side

Must close the end not in use, otherwise, other processes might accidentally write/read to it

dup2(): Use for duplication of file descriptors to known file descriptor

dup(): Same as dup2() but duplicates to random file descriptor

Sets new file descriptor to refer to old as well (i.e. old and new are the same now)

dup2(fd[0], STDIN_FILENO)	Redirect input to read from read end of pipe
dup2(fd[1], STDOUT_FILENO)	Redirect standard output to write to write end of pipe
dup2(file, STDOUT_FILENO)	Redirect standard output to write to contents of file

Unix Signals (IPC)

- Asynchronous notification regarding an event sent to a process/thread SIGKILL cannot be handled by user
  - E.g. kill, stop, continue
- Recipient of the signal must handle the signal via default handlers or user supplied handlers (signal(type, handler)) eg. signal(SIGSEGV, myfaulthandler)

Synchronisation

Race Conditions

- Execution outcome depends on the order in which the shared resource is accessed/modified (non-deterministic)
- General modification flow: load memory value into register, update register value, load register to memory
  - Race conditions happen when loading happens at the wrong time
- Caused by unsynchronized access to shared modifiable resources

Solution: designate code segment with race condition as critical section, only one process can execute in CS (synchronization)

Critical Section

```
// Normal code
Enter CS
// Critical work
Exit CS
// Normal code
```

Mutual exclusion: if process is executing in critical section, all other processes are prevented from entering the section

Progress: if no process is in a critical section, one of the waiting processes should be granted access

Bounded wait: after a process requests to enter the critical section, there exists an upper bound on how many other processes can enter the section before the process

Independence: process not executing in critical section should never block other process

Incorrect Synchronization

1. Deadlock: all processes blocked so no progress
2. Livelock: related to deadlock avoidance mechanism where processes keep changing state to avoid deadlock and make no other progress; processes are not blocked
3. Starvation: some processes never get to make progress in their execution as it is perpetually denied necessary resources

Test and Set Returns the old value of mem location before swapping TestAndSet(Register, MemoryLocation)

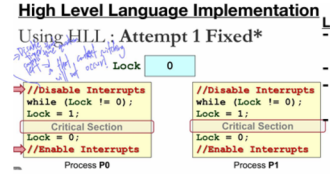
- Machine instruction provided by processors to aid synchronization
- Load the current content at MemoryLocation into Register and stores 1 into MemoryLocation (treated as lock)
- Performed as single machine operation (atomic)
- Used to create spinlocks

```
void EnterCS(int* Lock) {
  // Cannot enter if lock value is 1
  while (TestAndSet(Lock) == 1);
}
void ExitCS(int* Lock) {
  *Lock = 0;
}
```

Busy waiting: keep checking the condition until it is safe to enter critical section which is wasteful use of processing power

Variants: compare and exchange, atomic swap, load link/store conditional

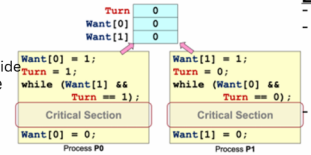
High Level Language Implementation



Limitations:

- Buggy CS can stall WHOLE system
- Use of busy waiting
- Requires permission to enable/disable interrupts
- Only works if both processors are on one CPU! (Because different CPU has different timer interrupts)

Peterson's Algorithm



- Assumption:
  - Writing to `Turn` is an atomic operation

Limitations

- Busy waiting
- Low-Level: Meaning it manipulates individual variables to achieve synchronization; more error prone; undesirable
- Not general (not easy to expand beyond 2 processes)

Example: Custom Signal Handler

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void myOwnHandler( int signo )
{
    if (signo == SIGSEGV) {
        printf("Memory access blows up!\n");
        exit(1);
    }
}

int main(){
    int *ip = NULL;

    if (signal(SIGSEGV, myOwnHandler) == SIG_ERR)
        printf("Failed to register handler\n");

    *ip = 123;

    return 0;
}
```

User defined function to handle signal. In this example, we handle the "SIGSEGV" signal, i.e. the memory segmentation fault signal.

Register our own code to replace the default handler.

This statement will cause a segmentation fault.

29

Eg of using shm functions: **shmid** = shmget(IPC_PRIVATE, 1 * sizeof(int), IPC_CREAT | 0600); **shm** = (int *) shmat(**shmid**, NULL, 0); **shm**[0] = 0 (doing shm[1] gives seg fault); shmdt((char *)**shm**); shmctl(**shmid**, IPC_RMID, 0);

NOTE: For shmat, 2nd arg is choosing where segment is attached and usually NULL because we want os to choose. third arg is permissions (0 is default).

technically u can disable & re-enable interrupts to act as universal lock since no quantum-based process switching so 1 proc running. However won't work on multi-core env since another process may enter CS on diff core. Generally disable interrupt = bad, may hang whole system