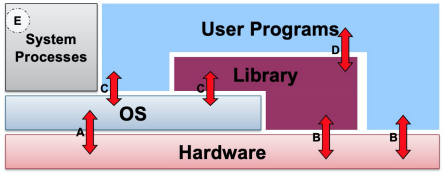
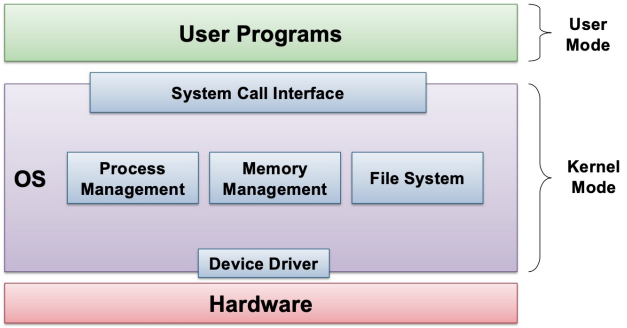
1. Intro to OS

- OS: program that acts as intermediary between use and hardware
- Motivation: manage resources and coordination, simplify programming, enforce usage policies, security and protection, user program portability, efficiency
- OS structure factors: flexibility, robustness, maintainability

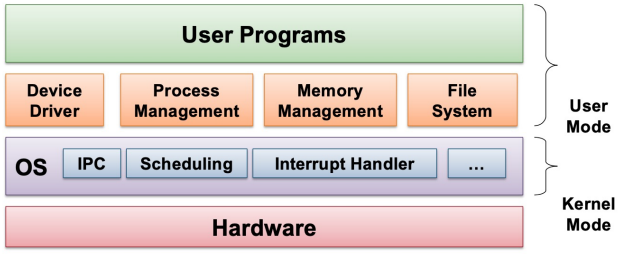


- **A**: OS executing machine instructions
- **B**: normal machine instructions executed (program/library code)
- **C**: calling OS using ***system call interface***
- **D**: user program calls library code
- **E**: system processes
  - Provide high level services, usually part of OS

- Monolithic OS: Most Unix variants, Windows NT/XP. Advantages: well understood, good performance. Disadvantages: highly coupled components, complicated internal structure

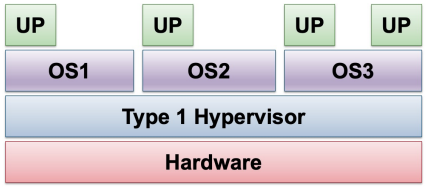


- Microkernel OS: kernel small and clean, basic functionalities (IPC, address space management, thread management), high level services build on top of basic / run as server process outside OS / communicate with IPC. Advantages: kernel robust and extendible, better isolation between kernel and high level services. Disadvantages: lower performance

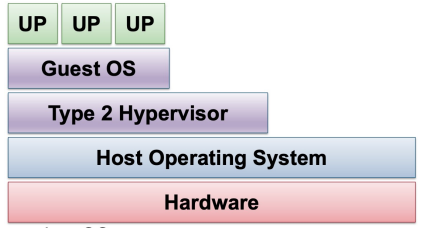


Generic Architecture of Microkernel OS Components

- Layered system is generalization of monolithic, client-server model is variation of microkernel
- Virtual machine / hypervisor: software emulation of hardware, virtualization of underlying hardware, primitive OS runs on top of VM
- Type 1 Hypervisor: provides individual VM to guest machines, egIBM VM/370

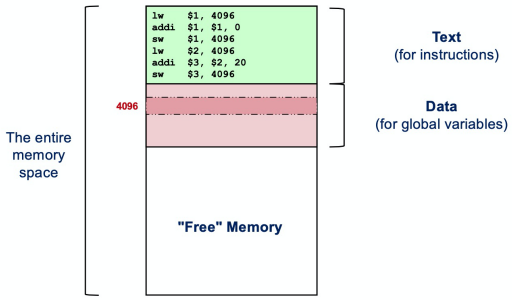


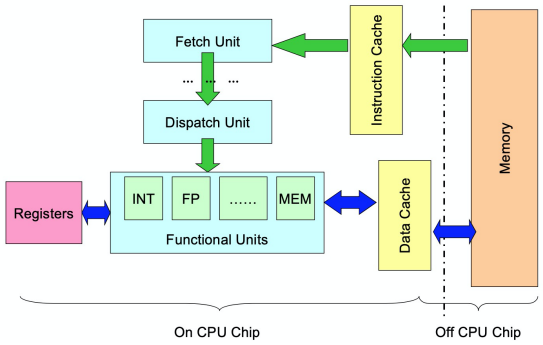
- Type 2 Hypervisor: runs in host OS, guest OS runs in VM. eg VMWare, VirtualBox



2. Process Abstraction

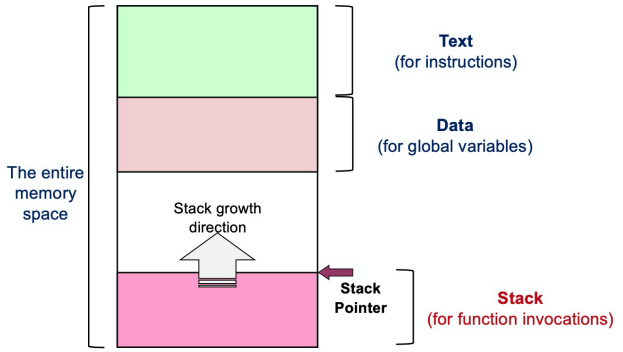
- Process: abstraction to describe running program
- Memory context: code, data. Hardware context: registers, PC. OS context: process properties, resources used

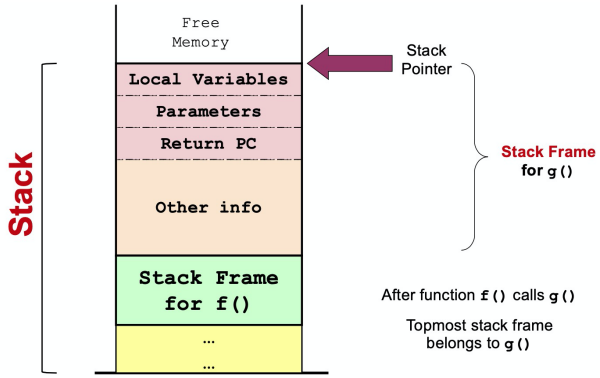


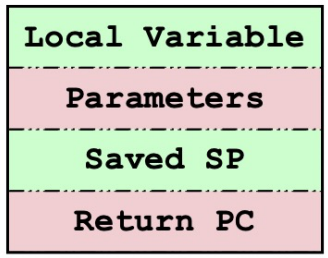


- Memory: storage for instruction and data. Cache: duplicate of memory for fast access, split into instruction cache + data cache. Fetch unit: loads instruction from memory, location indicated by PC (special register). Functional units carry out instruction execution. Registers: internal storage for fastest access, GPR accessible by user programs, special registers such as Program Counter (PC) and status register
- Function call steps: setup parameters, transfer control to callee, setup local variable, store result, return to caller

- Stack memory: memory region to store information needed for function invocation. Contains return address, parameters, storage for local variables, etc
- Top of stack region indicated by stack pointer (SP). Most CPU has specialized register for this purpose. Some memory layout may be flipped (stack on top, text on bottom)

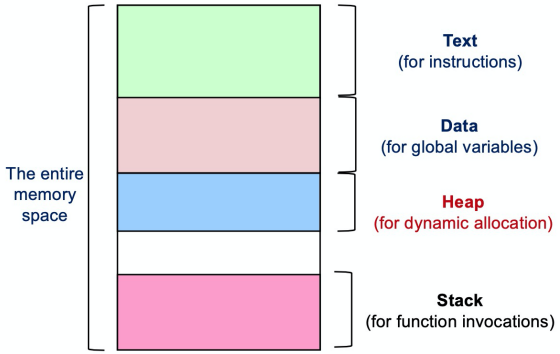




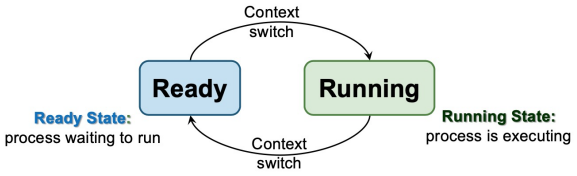


- Stack frame setup: prepare to make function call (caller pass parameters with registers and / or stack, caller save return PC on stack), transfer control from caller to callee (save registers used by callee, callee save old SP + FP, caller allocate space for local variables, callee adjust new SP to top of stack)
- use sw for saving registers
- Stack frame teardown: return from function call (callee restore saved registers, callee place return result on stack, callee restore FP +SP), transfer control back to caller using saved PC (caller utilizer result, continue execution in caller)
- Frame pointer (FP): stack pointer hard to use as it can change, FP points to fixed location in stack frame. Usage of FP is platform dependent.

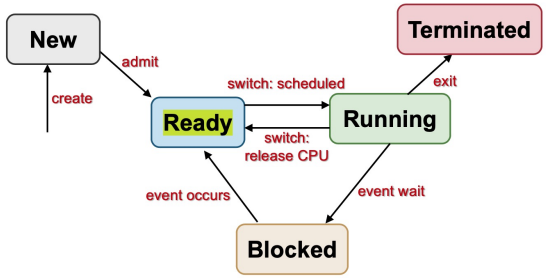
- Register spilling: use memory to temporarily hold GPR value, GPR can store other values, GPR can restore later
- Dynamically allocated memory: allocated at runtime, size not known at compile time, no definite deallocation timing. malloc() in C.



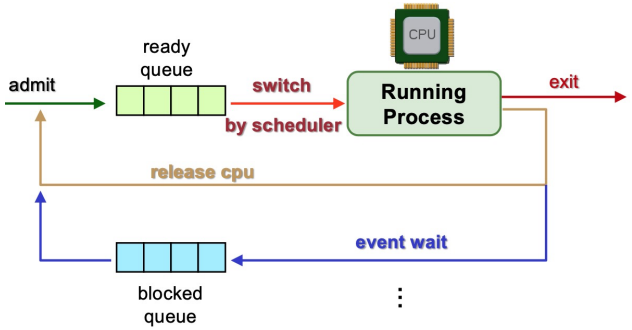
- Process model state diagram



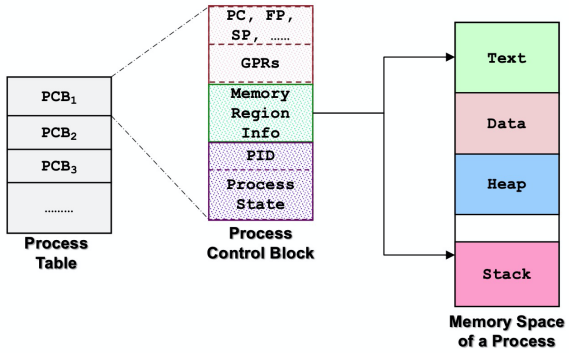
- Process model: set of states and transitions, describes behaviors of a process
- Generic 5-state process model:



- New: process newly created, may be still under initialization.
- Ready: process ready to run
- Running: process executed on CPU
- Blocked: process waiting for event, cannot execute until event is available
- Terminated: process finished execution, may need OS cleanup
- Create(nil -> new): new process is created
- Admit(new -> ready): process ready to be scheduled
- Switch(ready -> running): process selected to run
- Switch(running -> ready): process gives up CPU or preempted by scheduler
- Queuing model of 5-state transition

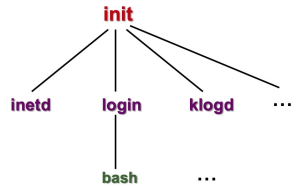


- PCB

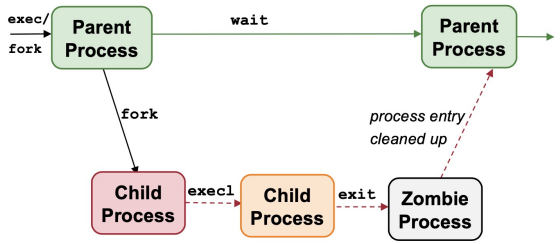


- System calls: API to OS, provides way of calling services in kernel, have to change from user mode to kernel mode
- Same name and same parameter syscalls: function wrapper, eg getpid(). Less parameters, flexible values: function adapter, eg printf()
- Syscall mechanism: User invokes library call, library places syscall number in register, library call executes special instruction (TRAP) to switch from user mode to kernel mode. In kernel mode: appropriate syscall number is determined using syscall number as index (handled by dispatcher), carry out request, switch from kernel to user mode, library call return to user program
- Exception: synchronous, eg divide by 0. Effect: execute exception handler, similar to forced function call
- Interrupt: asynchronous, eg timer, ctrl C. Effect: program execution suspended, execute interrupt handler.
- ps: Unix command for process information. Identification: PID. Information: process state (running, stopped, sleeping, zombie), parent PID, cumulative CPU time.
- int fork(): creates duplicate of current executable, returns PID of new process to parent / 0 to child. Child differs in PID, PPID, return value
- int execl(const char *path, const char *arg0, ...const char *argN, NULL): replace current executable with new one. PID and other information still the same.

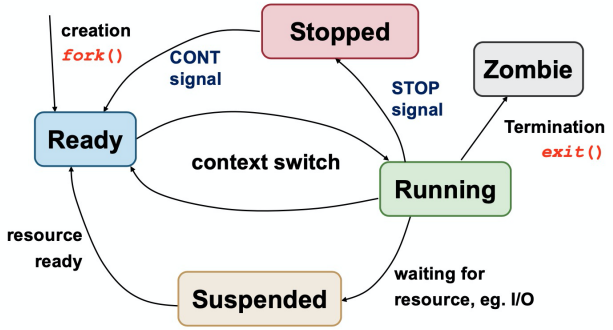
- init process: created at kernel at boot-up time, PID = 1



- void exit(int status): end execution of process. Status is 0 for normal, not 0 for problematic execution. Does not return.
- Most system resources are released on exit, eg file descriptors. Basic process resources not releasable, eg PID, status (for parent child synchronization), CPU time. PCB may still be needed.
- int wait(int *status): returns PID of terminated child. Status stores exit status of child, use NULL if don't need. Blocking, cleans up child system resources.
- Process interaction in Unix



- Parent terminates before child: init becomes pseudo parent, child sends termination signal to init, init calls wait() on child. Child terminates but parent did not call wait(): child becomes zombie, cannot be killed, can fill up process table, need to reboot.
- getpid(), getppid(): get process information
- Process state diagram in Unix

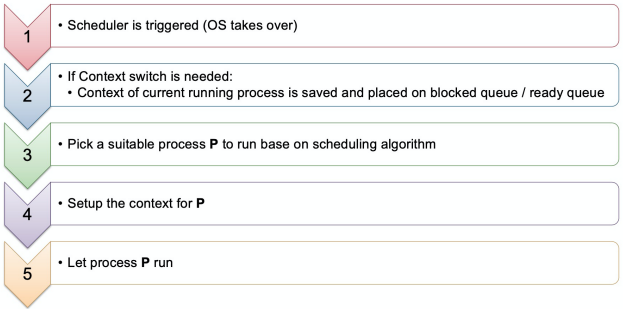


- implementing fork(): create address space of child process, allocate new pid, create kernel process data structure (process table entry), copy parent process' kernel environment (priority for scheduling), initialize child process context (pid, ppid, zero CPU time), copy memory regions from parent (program, data, stack, very expensive), acquire shared resources (file, current working directory), initialize hardware context (copy registers from parent), ready to run (add to scheduler queue)
- Copy on write: only duplicate memory location when written, else parent and child share same memory location

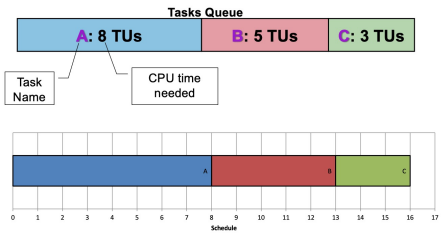
- Linux clone(): more versatile than fork(), can partial duplicate memory locations

3. Process Scheduling

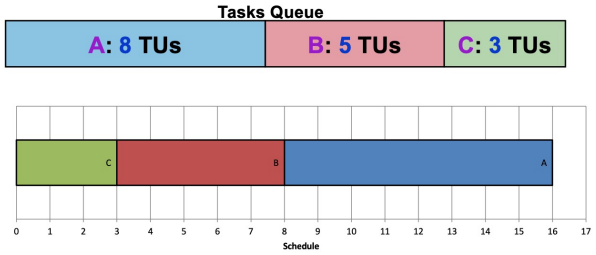
- Concurrent process: covers multitask process, could be virtual / physical parallelism
- Concurrent execution on 1 CPU interleaves instructions from both processes (time slicing)
- Context switch: multitasking to change context between two processes incurs overhead
- Scheduler: OS component that makes scheduling decision
- Scheduling algorithm: algorithm used by scheduler
- Typical process goes through CPU activity (computation, Compute-Bound process spend most time here), IO activity (request / receiver service from IO devices eg printing, read from file, IO-Bound process spend most time here)
- Processing environment: batch processing (no user, no interaction, no responsive), interactive (responsive and consistent), real-time (deadlines)
- Criteria: fairness (no starve, per-process or per-user), balance (all parts of computing system should be utilized)
- Scheduling policies: non-preemptive (process running until blocks or give up CPU), preemptive (given fixed time quota to run, when ended running process is suspended)



- Batch processing: FCFS, SJF, SRT
- No time quantum for batch processing
- Criteria for batch processing: turnaround time (finish - arrival time), throughput (number of tasks finish per time unit), CPU utilisation (percentage of time when CPU works on a task)
- First Come First Serve (FCFS): tasks stored in FIFO queue, pick first task in queue, run until task end or blocked. Blocked task removed from queue, placed at back of queue. Guaranteed no starvation. No responsiveness



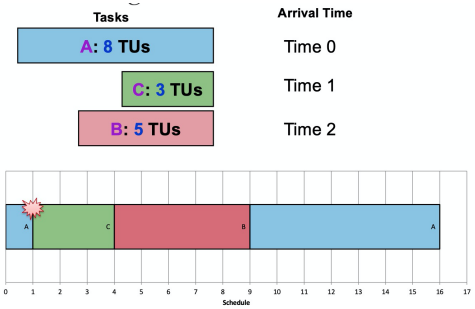
- Shortest Job First (SJF): select task with smallest total CPU time. Need to know total CPU time for task in advance, starvation is possible.



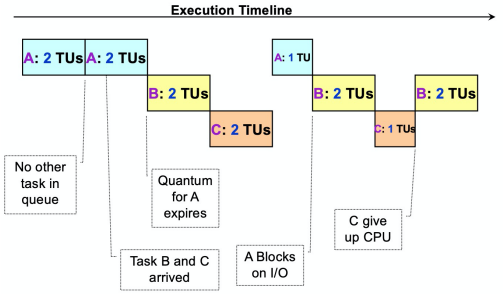
$$Predicted_{n+1} = \alpha Actual_n + (1-\alpha) Predicted_n$$

- $Actual_n$ = The most recent CPU time consumed
- $Predicted_n$ = The past history of CPU Time consumed
- α = Weight placed on recent event or past history
- $Predicted_{n+1}$ = Latest prediction

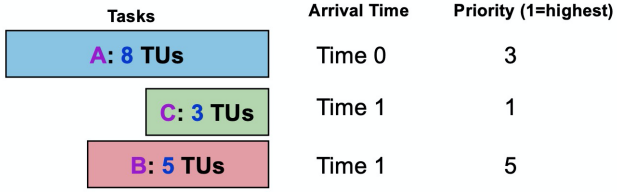
- Shortest Remaining Time (SRT): pre-emptive variation of SJF, select job with shortest remaining time. Can starve.

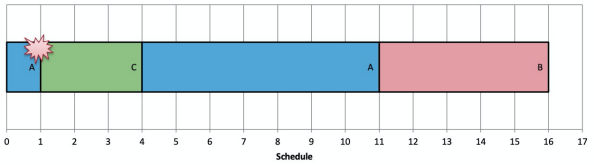


- Scheduling for interactive systems: RR, MLFQ, Priority, Lottery
- Criteria for interactive: response time, predictability.
- Preemptive scheduling algorithms used to ensure good response time.
- Timer interrupt: interrupt that goes off periodically, OS ensure timer interrupt cannot be intercepted by any other program, timer interrupt invokes scheduler
- Interval of Timer Interrupt (ITI): OS scheduler triggered every timer interrupt, typical values 1ms - 10ms
- Time Quantum: execution duration given to a process, could be constant / variable, must be multiple of timer interrupt, 5ms - 100ms
- Round Robin (RR): tasks stored in FIFO queue, pick first task from queue until task gives up CPU voluntarily / blocks / time quantum elapsed, task placed at back at queue (blocked task moved to other queue), blocked task placed at end of queue if ready again. Preemptive version of FCFS, response time guarantee, timer interrupt needed, choice of time quantum important (big = better CPU utilization but longer wait, small = big overhead but shorter waiting time)

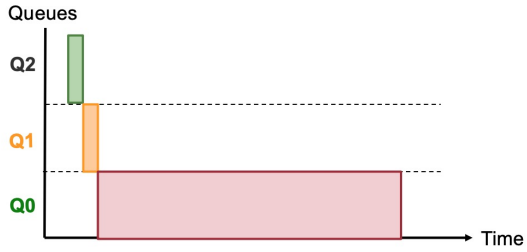


- Priority Scheduling: assign priority to processes, select task with highest priority. Preemptive: high priority process can preempt running process with low priority. Non-preemptive: high priority processes wait next round of scheduling. Can starve, worse in preemptive variant. Possible solution: decrease priority of currently running process after every time quantum / give current running process a time quantum and not consider it in next round of scheduling





- Priority inversion: A: high, B: medium, C: low. C arrives, locks resource, B preempts C, C still unable to unlock file, A arrives and need same resource as C, B runs first even if A has higher priority
- MLFQ: minimize response time for IO bound process, minimize turnaround time for CPU bound process. Rules: if priority(A) > priority(B), A runs. If priority(A) == priority(B), A and B runs in RR. New job has highest priority, job fully utilized time quantum, priority decreased. Job gives up / finish earlier, priority retained.

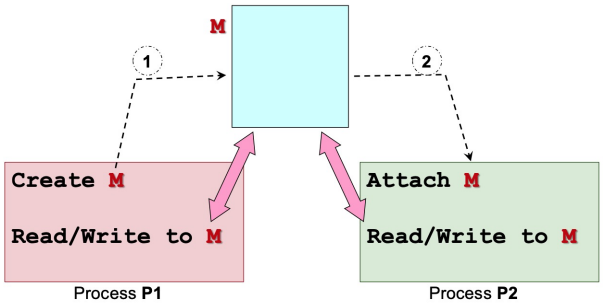


- Can abuse: A process comes in and always gives up before time slice, always higher priority. To fix: use cumulative CPU time. A process also can create fork and all child has high priority. To fix: child inherit parent's priority

- Lottery Scheduling: gives lottery tickets to process for various system resources eg CPU time, IO time. When scheduling decision needed, lottery picked randomly. Responsive (new process can participate in next lottery). Good level of control (process can distribute tickets to child, important process can have more tickets, different proportion of tickets per resource per task). Simple implementation.

4. Inter Process communication (IPC)

- Common IPC: shared memory, message passing
- Unix IPC: pipe, signal
- Shared Memory: communication via read / write to shared variables
- Idea: Process P1 creates a shared memory region M. P2 attaches M to its own memory space. P1 and P2 communicate using memory region M.



- OS only involved in step 1 and 2
- Shared Memory advantages: only create and attach involves OS, easy to use.
- Shared Memory disadvantages: needs synchronization (data races), harder implementation
- Basic steps: create shared memory region M, attach M to process memory space, read / write from M, detach M from memory space after use, destroy M (only one process does this, allows if M not attached to any other processes)

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
    int shmid, i, *shm;

    shmid = shmget( IPC_PRIVATE, 40, IPC_CREAT | 0600);

    if (shmid == -1){
        printf("Cannot create shared memory!\n");
        exit(1);
    } else
        printf("Shared Memory Id = %d\n", shmid);

    shm = (int*) shmat( shmid, NULL, 0 );
    if (shm == (int*) -1){
        printf("Cannot attach shared memory!\n");
        exit(1);
    }
```

```c
    shm[0] = 0;

    while(shm[0] == 0){
        sleep(3);
    }

    for (i = 0; i < 3; i++){
        printf("Read %d from shared memory.\n", shm[i+1]);
    }

    shmdt( (char*) shm);
    shmctl( shmid, IPC_RMID, 0);

    return 0;
}
```

The first element in the shared memory region is used as "control" value in this example (0: values not ready, 1: values ready).

The next 3 elements are values produced by the worker program.

Step 4+5. Detach and destroy Shared Memory region.

```c
//similar header files
int main()
{   int shmid, i, input, *shm;

    printf("Shared memory id for attachment: ");
    scanf("%d", &shmid);

    shm = (int*)shmat( shmid, NULL, 0);
    if (shm == (int*)-1) {
        printf("Error: Cannot attach!\n");
        exit(1);
    }

    for (i = 0; i < 3; i++){
        scanf("%d", &input);
        shm[i+1] = input;
    }
    shm[0] = 1;

    shmdt( (char*)shm );
    return 0;
}
```

Step 1. By using the region id directly, we in this case.

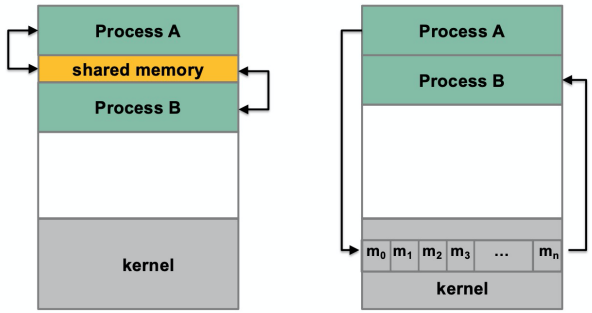
Step 2. Attach to sha memory region.

Write 3 values into

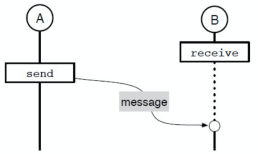
Let master program

Step 4. Detach Shar region.

- Message passing: explicit communication through message exchange.
- Idea: P1 prepares message, sends to P2, P2 receives message. Sending and receiving are usually syscalls
- Message have to be stored in kernel memory space
- All send / receive operations must go through OS (syscalls)
- Direct communication: sender / receiver explicitly name other party. Send (p2, message), Receive(p1 message). One link per pair of communicating processes, need to know identity of other party
- Indirect communication: messages send from message storage (mailbox / port). Send(mailbox, message), Receive(mailbox, message)

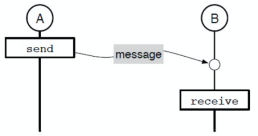


- Synchronous (blocking) primitives: send(): sender is blocked until message receives. receive(): Receiver blocked until message arrives.
- Asynchronous(non-blocking) primitives: send(): sender resume operation immediately. receive(): receiver receive message if available or indication that message is not ready.
- Blocking receive: most common, receiver must wait for message if not available.
- Non blocking receive: checks if message available, if available, retrieve and continue, else continue without message
- Non-blocking send: asynchronous. Sender never blocks, buffers the message. Disadvantage: too much freedom for programmer, program complex, finite buffer size -> sender has to wait when buffer full / returns error)



Scenario 1:

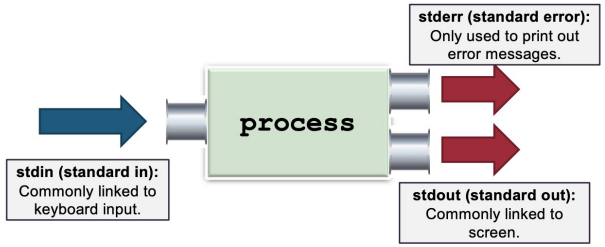
Message sent after `receive()` call.
Sender always continues.
Receiver is blocked waiting for message



Scenario 2:

Message sent before `receive()` call.
Sender always continues.
`receive()` gets data immediately

- Message buffers: intermediate buffers between sender and receive in asynchronous communication. Under OS control, no need synchronize. Large buffer decouples sender and receiver -> less sensitive to execution and do not wait for each other unnecessarily, zero buffering helps when sender faster than receiver. User needs to declare in advance mailbox size. Sender blocks if buffer full / returns error.
- Synchronous message passing: advantages: portable (easily implemented on different processing environment), easier synchronization (sender and receiver implicitly synchronized when synchronous primitive used). Disadvantages: inefficient (needs OS), harder to use (message limited in size / format)
- Unix pipes: 3 default communication channels. In C: printf() uses stdout, scanf() uses stdin. Earliest IPC mechanism.



- Use "|" to link input / output channels, known as piping. Output of A goes to input of B (as keyboard).
- Pipe shared between two processes in the form of producer-consumer. P produces (writes) n bytes, Q consumes (reads) m bytes. Behaves like anonymous files, FIFO.
- Pipe functions as circular bound byte buffer with implicit synchronization: writer writes when buffer full, readers wait when buffer empty

- Variants: multiple readers / writers, half-duplex (one direction for read, one for write), bidirectional
- Unix pipe syscall: int pipe(int fd[]), returns 0 for success, !0 for errors. fd[0] is reading end, fd[1] is writing end


```c
#define READ_END 0
#define WRITE_END 1

int main()
{
   int pipeFd[2], pid, len;
   char buf[100], *str = "Hello There!";

   pipe( pipeFd );
   if ((pid = fork()) > 0) { /* parent */
     close(pipeFd[READ_END]);
     write(pipeFd[WRITE_END], str, strlen(str)+1);
     close(pipeFd[WRITE_END]);
   } else {                  /* child */
     close(pipeFd[WRITE_END]);
     len = read(pipeFd[READ_END], buf, sizeof buf);
     printf("Proc %d read: %s\n", pid, buf);
     close(pipeFd[READ_END]);
    }
}
```

- Possible to change standard communication channels (stdout, stderr, stdin) to one of the pipes
- Unix signal: asynchronous notification regarding an event. Recipient of signals must handle signal by default handlers / user supplied handler (only some, except kill -9)
- Common signals: kill, stop, continue, segmentation fault

```c
#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;
}
```

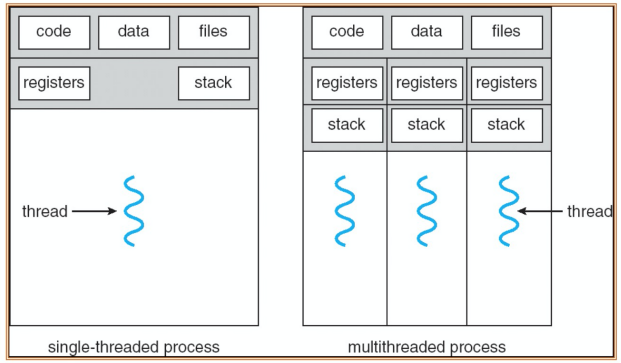
User defined func
In this example, v
"SIGSEGV" signa
segmentation fau

Register our
the default ha

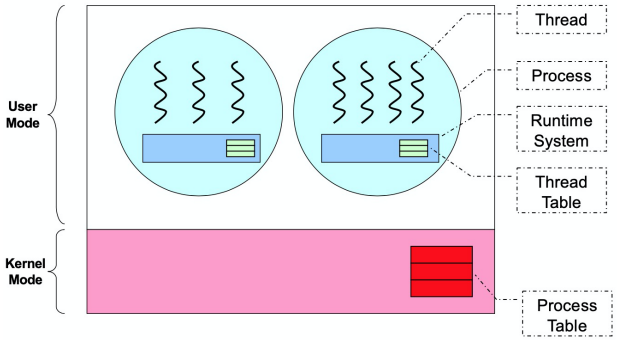
This statement will cause a segmentation fault.

5. Threads

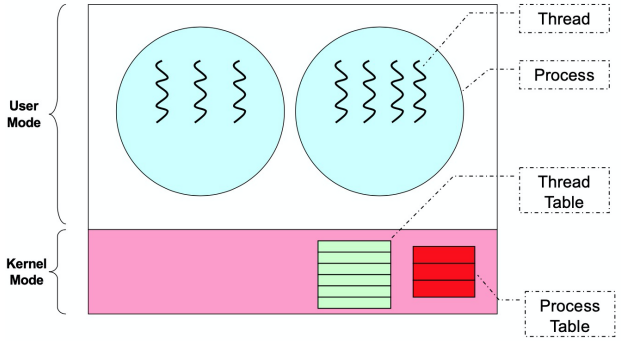
- Motivation: process creation with fork() is expensive (duplicates most memory space, process context etc). Context switch requires saving / restoring of process information. Hard for process to communicate with each other (need IPC)
- Idea: traditional process has one thread of control (only one instruction executing all the time), add more threads to control same process (multiple parts of program is executing same time)
- A single process can have multiple threads, known as multithreaded process
- Threads in same process share Text, Data, Heap, process ID but not thread id (identification), registers, stack



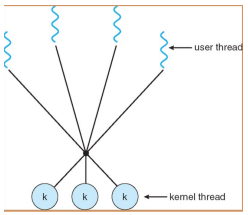
- Process context switch requires OS context, hardware context, memory context. Thread switch only requires hardware context (registers, stack - changing FP and SP)
- Thread benefits: economy, resource sharing, responsiveness, scalability
- User thread: thread implemented as user library, kernel cannot see user threads. Advantages: can multithread on any OS, thread operations are just library calls, configurable and flexible. Disadvantages: OS not aware of threads, scheduling performed as process level (one thread blocks = process blocks = all threads block), cannot exploit multiple CPU



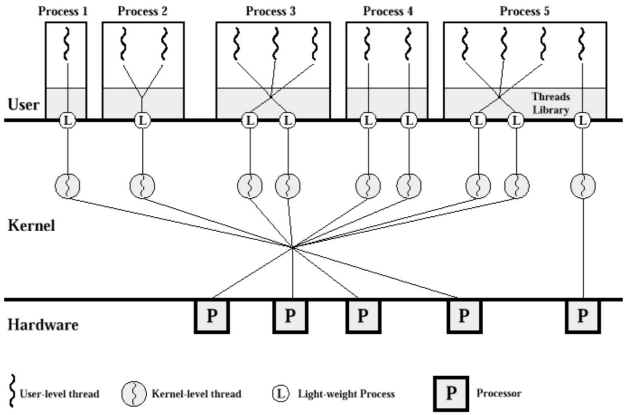
- Kernel threads: thread implemented in the OS, thread operation handled as syscalls, thread-level scheduling is possible (kernel schedule by threads instead of process), kernel may make use threads for its own execution. Advantages: kernel can schedule on thread level (more than 1 thread can run simultaneously on multiple CPU). Disadvantage: thread operations are sys calls (slower, resource intensive), less flexible (used by all multithreaded programs)



- Hybrid thread model: OS schedule on kernel threads only, user thread bind to kernel thread. Great flexibility, can limit concurrency of any process



- Solaris hybrid thread model



- Threads started as software mechanism, have hardware support on modern processors (supply multiple sets of registers eg GPR to allow threads run natively and parallel on same core), known as simultaneous multi-threading (SMT), eg hyperthreading on Intel processor
- pthread creation: tidCreated = threadId for created thread, threadAttributes = control the behavior of new thread, startRoutine = function pointer to the function to be executed by thread, argForStartRoutine = arguments for startRoutine function

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

- pthread termination: exitValue = Value to be returned to whoever synchronize with this thread. If pthread_exit() not used, pthread will terminate automatically at the end of startRoutine. Return value captured as exitValue. Else. exitValue not defined.

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

- Example of creation and termination

```
//header files not shown

void* sayHello( void* arg )          // Function to be executed by a pthread
{
    printf("Just to say hello!\n");
    pthread_exit( NULL );            // Pthread Termination
}

int main()
{
    pthread_t tid;                   // Pthread Creation

    pthread_create( &tid, NULL, sayHello, NULL );
    printf("Thread created with tid %i\n", tid);

    return 0;
}
```

- pthread synchronization: join. Returns 0 if success, !0 otherwise. threadID = TID of pthread to wait for, status = exit value returned by target pthread

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

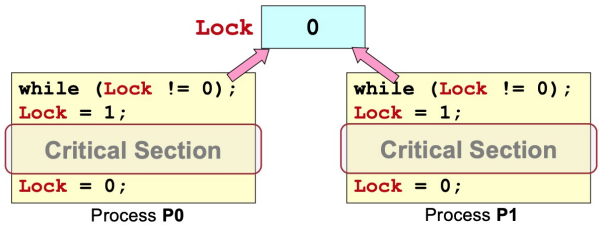
6. Synchronization

- Condition for race condition: two or more processes execute concurrently in interleaving fashion and share modifiable resource, instructions are not atomic
- Execution of single sequential process is deterministic, concurrent process is non-deterministic (race conditions)
- Critical section: designate code segment with race condition. At any point, only one process can execute in it

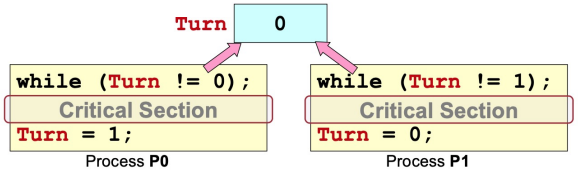
```
        //Normal code
Enter CS
        //Critical Work
Exit CS
        //Normal code
```

Critical Section

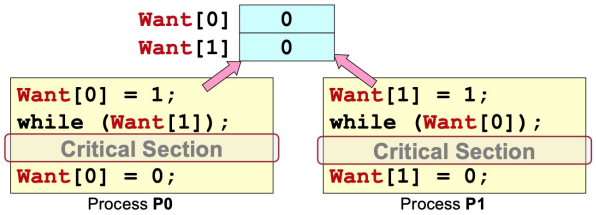
- Properties of correct CS implementation: mutual exclusion (if process P1 is executing in CS, all other processes are prevented from entering), progress (if no process in CS, one of the waiting progress must enter), bounded wait (after P1 request to enter CS, there exist a limit of times other processes can enter before P1), independence (processes not execute in CS should not block other process)
- Incorrect synchronization: deadlock (all progresses block), livelock (processes avoid deadlock and no progress, typically not blocked), starvation
- No mutual exclusion:



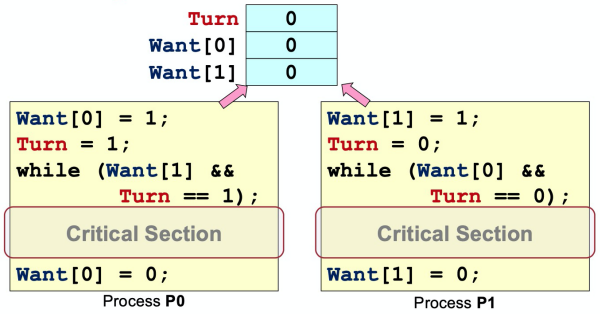
- Preventing context switch: if process hangs will stall whole system, busy waiting, requires permission to disable / enable interrupts
- no independence (if p0 never enters, p1 starve)



- Deadlock (if p1 wants to enter, give turn to p0, vice versa)



- Peterson's algorithm: complex, hard to generalize to multiple processes, busy waiting

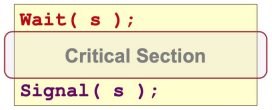


- TestAndSet: load and store is atomic (1 machine operation). Disadvantage: busy waiting

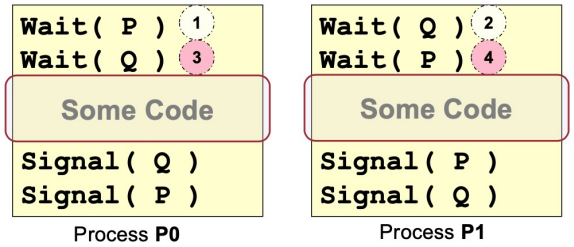
```
void EnterCS( int* Lock )
{
   while( TestAndSet( Lock ) == 1);
}
```

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

- Semaphore: a semaphore S has an integer value, initialized to non-negative. wait(S): if S <= 0, blocks. Decrements S, also known as P() or Down(). signal(S): S++, wakes processes, never blocks, also known as V() or Up()
- Given $S_{initial} >= 0$, $S_{current} = S_{initial}$ + number of signals executed + number of wait completed
- General semaphores (counting semaphores): S = (0, 1, 2, 3, …)
- Binary semaphores (mutex): S = 0 or 1
- General semaphores provided for convenient
- General semaphores can be implemented with binary and vice versa
- Critical section and semaphore: binary semaphore S = 1



- Mutex correctness proof: N_{CS} = number of processes in CS = process that completed wait() but not signal() = Wait(S) - Signal(S). $S_{initial} = 1$, $S_{current} = 1$ + Signal(S) - Wait(S), $S_{current} + N_{CS} = 1$, since $S_{current} >= 0$, $N_{CS} >= 1$
- Deadlock proof: all process stuck at wait(S), $S_{current} = 0$ and $N_{CS} = 0$, but $S_{current} + N_{CS} = 1$, contradiction.
- Starvation proof: p1 blocked at wait(S), means p2 is in CS, exits with signal(S), p1 eventually wakes up
- Incorrect use of semaphore: deadlock. Assume P = 1, Q = 1



- Dining philosopher (Tanenbaum):

```
#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 );
}
```

- Dining philosopher (limited eater): at most 4 eaters

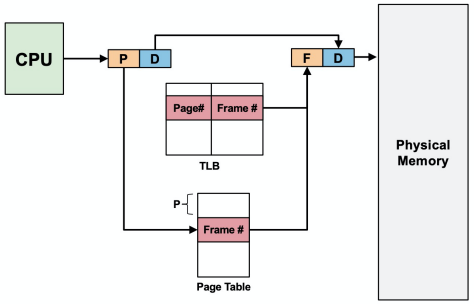
```
void philosopher( int i ){
    while (TRUE){
       Think( );
       wait( seats );
       wait( chpStk[LEFT] );
       wait( chpStk[RIGHT] );
       Eat( );
       signal( chpStk[LEFT] );
       signal( chpStk[RIGHT] );
       signal( seats );
    }
}
```

7. Memory Management

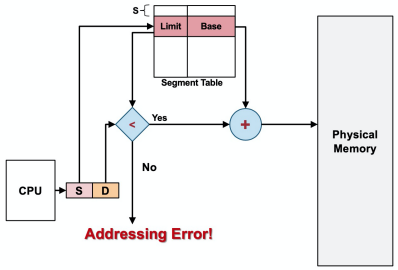
- Without memory management: no mapping required, fixed address in compilation, faster, but conflict if two processes use same memory address
- Address relocation: add another offset to all memory references in second process, but slow loading time, hard to distinguish memory reference from normal integer
- Base + limit: slow (need add and compare), generalized to segmentation
- Supports multitasking (allow multiple processes in the physical memory at the same time), free memory by removing terminated processes / swap blocked processes to secondary storage
- Fixed size partition: easy to allocate, easy to manage (allocate to first one available, no need choose), but partition size need to be large enough, internal fragmentation when small process waste memory space
- Variable Size / dynamic partition: no internal fragmentation, but need to maintain more information in OS, takes more time to locate appropriate region, external fragmentation when process creation / termination / swapping
- First Fit: fastest, Best Fit: best memory utilization, Worst Fit: low production of holes
- Buddy system: two blocks B and C are buddies of size 2^S if the Sth bit of B and C is complement, leading bits up to Sth bit are the same

8. Disjoint Memory Schemes

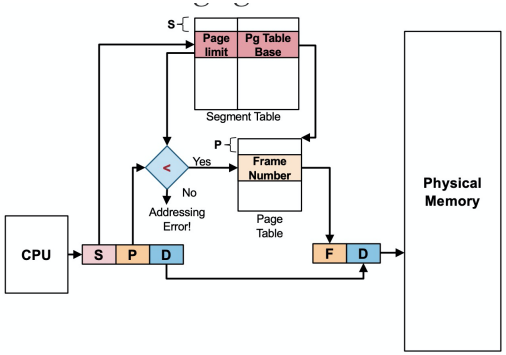
- Physical memory splits into physical frames, logical memory splits into pages of the same size. Logical memory contiguous, frames disjoined.
- Physical address = physical frame number (get from page table with page number) x size of physical frame + offset
- Given page/frame size of 2^n, m bits of logical address, p = most significant (m - n) bits of address, d = remaining n bits, use p to find frame number f. Physical address = f*(2^n) + d
- Paging removes external fragmentation but still have internal fragmentation
- TLB: hardware support, a cache of a few page table entries. Use page number to search TLB, if entry found (TLB-hit), generate physical address. Else (TLB-miss), access full page table in memory, retrieve frame number, update TLB entry.



- Memory access time = p(TLB hit) * (TLB access time + memory access time) + p(TLB miss) * (TLB access time + memory access time + memory access time)
- When context switch, TLB entries flushed.
- Access right bits: several bits attached to PTE to determine if it is writable, readable, executable.
- Valid bits: several bits attached to PTE to check for out of range (caught by OS)
- Page sharing: processes share same memory frame, implement copy-on-write (parent and child share same PTE until writing happens, create new frame and update child PTE)
- Segmentation: separate memory into code, data, text, stack. Each memory segment has a name (ID) and a limit, specified as segment name + offset
- Segment ID used to look up the base and limit of a segment in the segment table. Physical address = base + offset, and offset has to < limit.

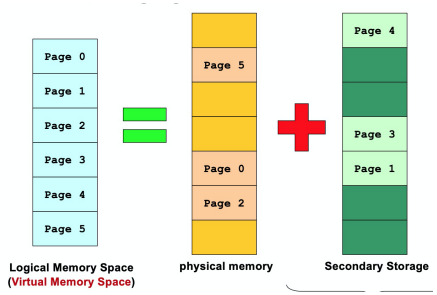


- Segmentation: each segment can grow / shrink / protect / shared independently, but can cause external fragmentation
- Segmentation with paging: each segment has a page table

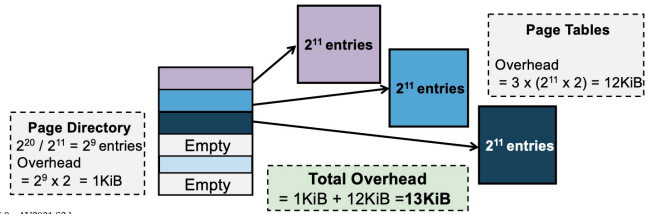


9. Virtual Memory Management

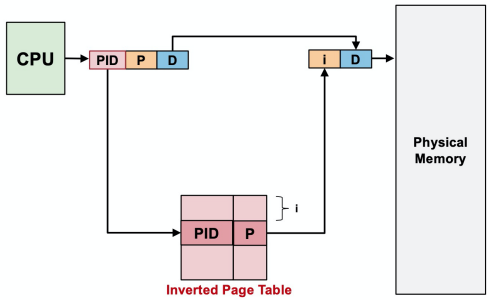
- Logical memory splitted into pages, some pages in secondary memory, some in physical memory



- Memory resident bit: check if page in physical memory or (resident) or secondary memory (non-resident)
- Page fault: CPU tries to access non-resident page, OS needs to bring non-memory resident page into physical memory.
- Accessing page steps: is page memory resident? -> access physical memory location, else -> page fault, trap to OS -> locate page X in secondary storage -> load page X into physical memory -> update page table -> re-execute same instruction (go to step 1)
- Thrashing: memory access results in page faults most of the time. Solvable by locality principles
- Temporal locality: memory address which is likely to be used again
- Spatial locality: memory address close to a used address likely to be used
- Direct paging: 2^p pages in logical memory space, need p bits to specify one unique page. Page table has 2^p PTE. Example, virtual address 32 bits, page size 4 KB = 2^12, p = 32 - 12 = 20, table has 2 ^ 20 entries. If PTE size is 2 bytes, page table size is 2 ^ 20 * 2 = 2 ^ 21 = 2 MB per process
- 2-level paging: split page table into smaller page tables (each with a page table number), use a directory to keep track of splitted pages. If the original page table has 2^p entries, with 2^m smaller page tables, m bits needed to uniquely identify each table, each smaller table contains 2^(p - m) entries. Page directory contains 2^m indices to locate each smaller page table.
- Example: virtual address 32 bits, physical memory 512 MB, page size 4 KB, PTE size 4 bytes, each process uses 512 MB. For direct paging, total number of PTE = 2^32 / 2^12 = 2^20, page table size per process = 2^20 * 2^2 = 2^22. For 2-level paging, number of PTEs per page tablet = page size / PTE size = 2^12 / 2^2 = 2^10, so number of page tablet needed for all PTEs = total PTE / number of PTEs per page tablet = 2^20 / 2^10, each process needs process size / page size = 2^29 / 2^12 = 2^17 pages, so each process needs 2^17 / 2^10 page tablets = 2^7 page tablets. Page directory size = 2^10 * 2^2, total memory = 2^10 * 2^2 + 2^7 * 2^10 * 2 ^ 2.
- Overhead for 2-level paging = 1 page directory + 3 smaller page tables



- Inverted page table: entries ordered by frame number. One table for all processes but slow translation.



- Page replacement: when no physical memory free during page fault, need to evict memory page. If dirty, need to write back to secondary, else no need.
- Place replacement algorithm: OPT, FIFO, LRU, CLOCK
- Memory access time: $(1 - p)* T_{mem} + p * T_{page\_fault}$
- OPT: replace page that will not be used again for the longest period of time. Pros: guarantees minimal page faults. Cons: not realizable as need future knowledge, but used as a baseline comparison algorithm

Time	Memory Reference	Frame			Next Use Time			Fault?
		A	B	C				
1	2	2			3	?	?	Y
2	3	2	3		3	9	?	Y
3	2	2	3		6	9	?	
4	1	2	3	1	6	9	?	Y
5	5	2	3	5	6	9	8	Y
6	2	2	3	5	10	9	8	
7	4	4	3	5	?	9	8	Y
8	5	4	3	5	?	9	11	
9	3	4	3	5	?	?	11	
10	2	2	3	5	12	?	11	Y
11	5	2	3	5	?	?	11	
12	2	2	3	5	?	?	?	

- FIFO: evicts oldest memory page. Pros: simple to implement. Cons: Belady's anomaly (page fault increases as frame increases because does not exploit temporal locality)

Time	Memory Reference	Frame			Last Use Time			Fault?
		A	B	C				
1	2	2			1			Y
2	3	2	3		1	2		Y
3	2	2	3		3	2		
4	1	2	3	1	3	2	4	Y
5	5	2	5	1	3	5	4	Y
6	2	2	5	1	6	5	4	
7	4	2	5	4	6	5	7	Y
8	5	2	5	4	6	8	7	
9	3	3	5	4	9	8	7	Y
10	2	3	5	2	9	8	10	Y
11	5	3	5	2	9	11	10	
12	2	3	5	2	9	11	12	

- LRU: replace page that has not been used in the longest time (temporal locality). Pros: good results, close to OPT. Cons: hard to implement, need hardware support (time forever increasing - overflow, need to search through stack entries if implemented with stack)

Time	Memory Reference	Frame			Last Use Time			Fault?
		A	B	C				
1	2	2			1			Y
2	3	2	3		1	2		Y
3	2	2	3		3	2		
4	1	2	3	1	3	2	4	Y
5	5	2	5	1	3	5	4	Y
6	2	2	5	1	6	5	4	
7	4	2	5	4	6	5	7	Y
8	5	2	5	4	6	8	7	
9	3	3	5	4	9	8	7	Y
10	2	3	5	2	9	8	10	Y
11	5	3	5	2	9	11	10	
12	2	3	5	2	9	11	12	

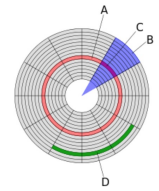
- CLOCK: modified FIFO. Each PTE has a reference bit: 1 = accessed, 0 = not accessed. Oldest FIFO page selected, if reference page = 0, page replaced. Else, reference bit = 0.

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)	

- Frame allocation simple approaches: equal allocation (each process gets N / M frames), proportional allocation (each process gets $size_p / size_{total} * N$) frames
- Local replacement: replaced page selected among pages of process that causes page fault. Pros: stable performance between multiple runs. Cons: if frame allocated is not enough, process' progress gets hindered.
- Global replacement: process P can take a frame from process Q by evicting Q's frame. Pros: allow self-adjustment among processes. Cons: badly-behave process can affect others, frames allocated different between multiple runs
- If global replacement is used, cascading thrashing may occur. If local replacement is used, the process can hog I/O and degrade the performance of other processes.
- Working set model W(time, interval): active pages at interval at a time

10. File System Introduction

- File system: abstraction on top of physical media, high level resource management, protection between processes and users, sharing between processes and users

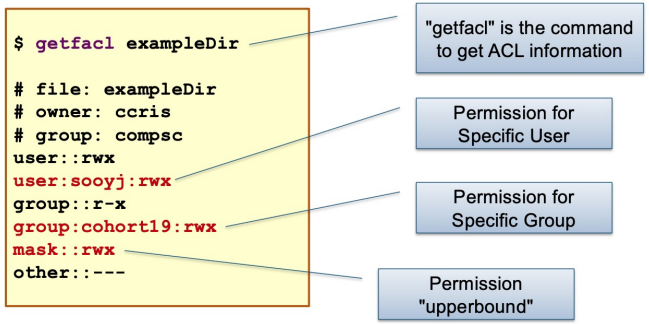


A = Track
B = Geometric Sector
C = Track Sector
D = Cluster

- General criteria: self-contained (plug-and-play), persistent (lifetime beyond OS), efficient (minimum overhead, good management of space)
- File: ADT that represents a logical unit created by process, contains data and metadata (file attributes)

Name:	A human readable reference to the file
Identifier:	A unique id for the file used internally by FS
Type:	Indicate different type of files E.g. executable, text file, object file, directory etc
Size:	Current size of file (in bytes, words or blocks)
Protection:	Access permissions, can be classified as reading, writing and execution rights
Time, date and owner information:	Creation, last modification time, owner id etc
Table of content:	Information for the FS to determine how to access the file

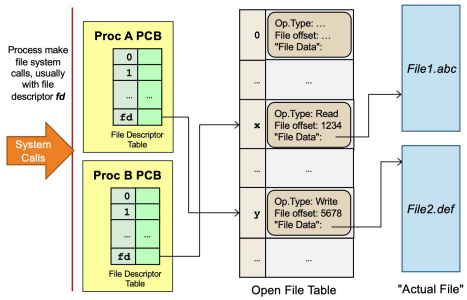
- Common file types: regular files (ASCII / text file, binary files eg pdf, mp3/mp4) , directories, special files (character/block oriented)
- Windows use extension to distinguish file type, Unix uses magic number
- Type of access: read, write, execute, append, delete, list
- Permission bits: owner (user who created file), group (same department), universe (all other users)
- Access control list



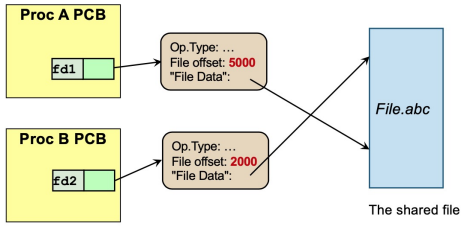
- Operations on file metadata: rename (change filename), change attributes (file access permissions, dates, ownership), read attribute (get file creation time)
- Sequential access: read in order from beginning
- Random access: read data in any order, read(offset) and seek(offset)
- Direct access: used for file contains fixed length record, allow random access to any record directly, useful when record amount is huge eg database
- File operations

Create:	New file is created with no data
Open:	Performed before further operations To prepare the necessary information for file operations later
Read:	Read data from file, usually starting from current position
Write:	Write data to file, usually starting from current position
Repositioning:	Also known as seek Move the current position to a new location No actual Read/Write is performed
Truncate:	Removes data between specified position to end of file

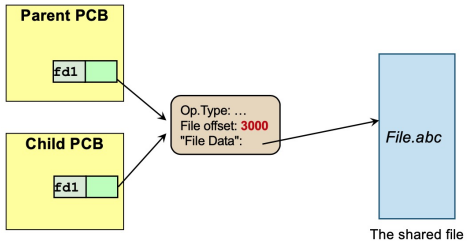
- Information kept for open file: file pointer, disk location, open count (how many processes open this file)



- Two processes using different file descriptors (two process opens same file / same file opened twice)



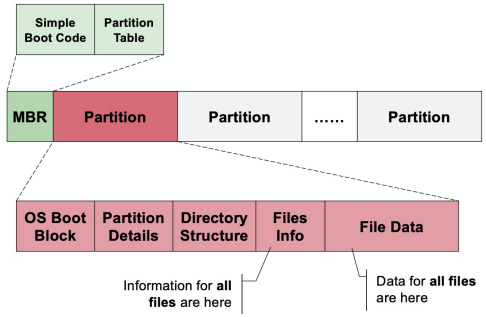
- Two processes using same file descriptors (fork)



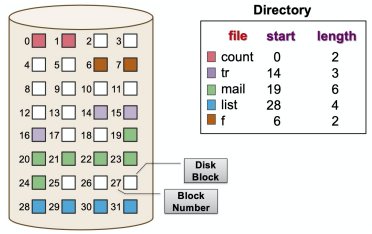
- Hard link: two directories A and B have separate pointers to actual file F in disk. Pros: low overhead, only pointers needed. Cons: deletion problems. Command: ln
- Symbolic link: B creates a special link file, G, which contains path name of F. When G accessed, find F, access F. Pros: simple deletion, if B deletes: G deleted, not F. If A deletes: F gone, G remains (but not working). Command: ln -s
- General graph: can be created in Unix (symbolic link to directory). Cons: hard to traverse (need to prevent infinite loop), hard to determine when to remove a file

11. File System Implementations

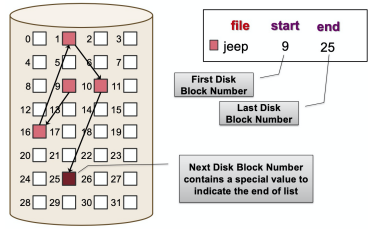
- Smallest accessible unit of disk structure: logical block (512 B to 4KB, mapped onto disk sectors)
- File system contents: OS-boot up info, partition details (total number of blocks, number and location of free blocks), directory structure, files information, actual file data



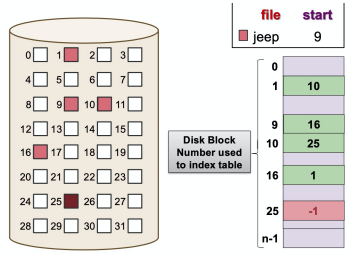
- When file size != multiple of logical blocks: internal fragmentation
- Contiguous: allocate consecutive disk blocks to a file. Pros: simple to keep track, fast access. Cons: external fragmentation (variable-sized partition), file size need to be specified



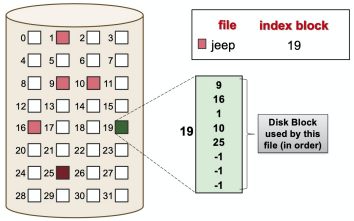
- Linked List 1: each disk block stores the next disk block counter, actual file data. File information stores the first and last disk block number. Pros: no fragmentation. Cons: random access in file is slow, part of disk block is used for pointer (5 KB file data need more than 5 disk blocks), less reliable (file pointers lost)



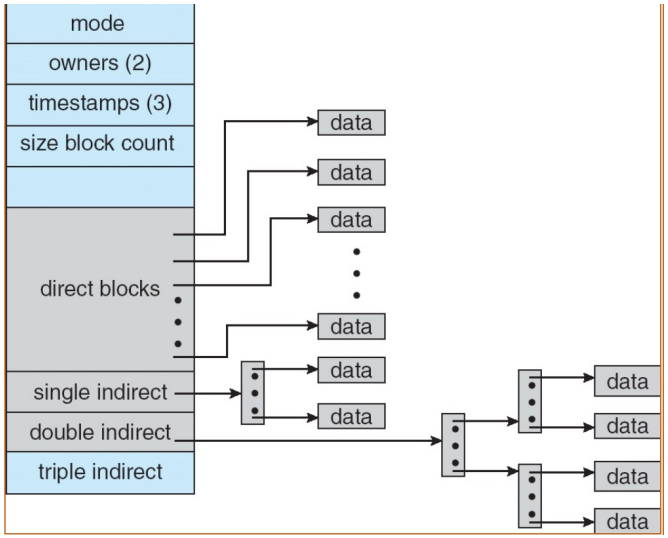
- FAT: move all block pointers into a single table, in memory all time, used by MS-DOS. Pros: faster random access (traversal now in memory, faster). Cons: FAT can be huge when disk large



- Indexed allocation: each file has an index block. Pros: less memory overhead (only index block of opened file needs to be in memory), fast direct access. Cons: index block overhead, limited max file size (max number of block = number of index block entries)



- Combination of direct indexing and multi-level index: Unix I-Node has 12 direct pointers that point to disk blocks directly, 1 single indirect block which contains a number of direct pointers, 1 double indirect block that points to a number of single indirect blocks, 1 triple indirect block.



- Free space management: maintain free space information
- Bitmap: each disk block represented by 1 bit, 1 = free, 0 = occupied. Pros: good set of manipulations (find the first free block, next n free blocks easily allocated). Cons: need to keep in memory for efficiency
- Linked list: each disk block contains a number of free disk block pointers, and a pointer to the next free space disk block. Pros: easy to locate free block, only first pointer need to be stored. Cons: high overhead, but can be solved by storing free block list in free blocks
- Directory structure: keep track of files in directory (with file metadata), map file name to file information
- Linear list: each entry represents a file (store file name / information / pointer to file information). Cons: slow finding / deep tree traversal
- Hash table: each directory contains a hash table, file name hashed into index. Pros: fast lookup. Cons: limited size hash table, need good hash function
- File information: file name, disk blocks and other metadata
- Create operation: use full pathname to locate parent directory -> search for filename to avoid duplicates -> use free space list to find free disk blocks -> add an entry to parent directory (file name, disk block info)
- Open operation: search system wide table for existing entry -> return pointer if found -> use full pathname to locate file -> load new entry into system wide table -> create entry in process table to point to entry -> return pointer to this entry

- I/O (disk) scheduling: focus on reducing seek time
- Disk scheduling algorithms: FCFS, SSF (shortest seek first), SCAN (elevator)
- SCAN

- New algorithms: Deadline (3 queues: sorted, read FIFO, write FIFO), no-operation (no sorting), cfq (completely fair queueing, time slice and per-process sorted queue), bfq (budget fair queueing, multiqueue: fair sharing based on number of sectors requested)