CS2106 Cheatsheet

What is OS?

- Program that acts as intermediary between computer user and hardware

→ Windows, Mac, Ubuntu, Solaris

→ iOS, Android

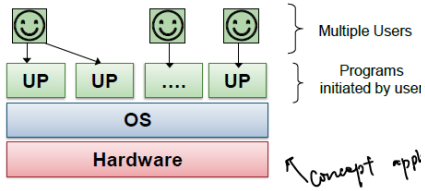
Mainframes

- No interactive interface, support batch processing only
- Simple batch processing is inefficient – CPU idle when performing I/O

→ To solve: Time-sharing OS, where multiple users use mainframe at the same time

Time-Sharing OS

- Allow multiple users to interact, uses job scheduling with context switching, OS manages sharing of CPU time, memory and storage



→ Hardware is virtualized, user program executes as if it has all the resources to itself

Motivations of OS

- Large variation in hardware configs, but hardware in same category has well defined, common functionality
- Abstraction → Present user with common high level functionality

→Efficient, programmable and portable

- Resource Allocator

→ Allow programs to execute simultaneously for better utilisation of resources

- Control program

→ Prevents users from improperly using computer, act as safety net if users are stuck in program with bug (infinite loop for e.g.)

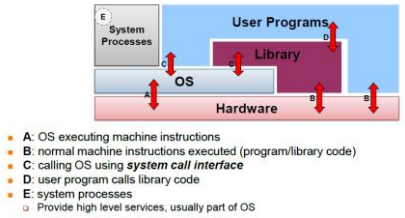
→ Enforce usage policies, improves security and protection

High Level view of OS

- Essentially a software that runs in kernel mode

→Kernel mode: has complete access to all hardware resources

→Other softwares execute in user mode, with limited access to hardware resources



→ OS is basically a program (kernel) that deals with hardware issues, provides system call interface

→Has special codes for interrupt handlers, device drivers etc

→Historically in assembly/machine code, now in HLLs like C++/C

→ Difficult to debug, high complexity to code, big codebase, no one to rely on for nice svcs

OS Structures

→ Monolithic, microkernel, layered, client-server, exo-kernel, hybrid

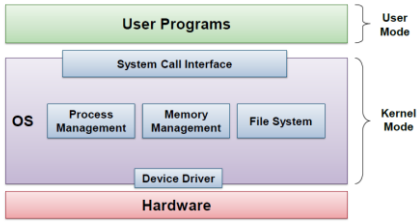
Monolithic OS

- Kernel is one big special program, just like a monolithic repository in SWE

→ Most unix variants, DOS, windows 9x

→ Good performance, well understood

→ Highly coupled (maybe), complicated internal structure

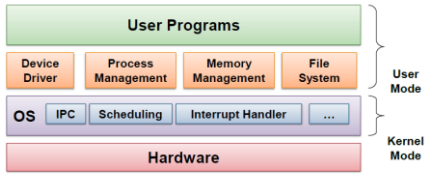


Microkernel OS

- Kernel is very small and clean, only provides basic and essential facilities: IPC (Inter-process communication), address space management, thread management
- Higher level OS services built on top of basic facilities, run as server process outside of kernel and uses IPC to communicate

→ More robust, more extensible, better isolation/protection between kernel and high level svcs

→ Less performance, have to interact through IPCs, whereas monoliths can use function calls



Generic Architecture of Microkernel OS Components

Other OS Structures

→ Layered Systems, a generalization of monoliths. Organises components into hierarchy of layers, where upper layers make use of lower layers, lowest layer = hardware, highest = UI

→ Client-Server Model, a variation of microkernel, client processes request service from a server process. Server process built on top of microkernel, client and server can be on separate machine

Virtual Machines

Motivation: Cloud computing, run several OSes on the same hardware, test potentially destructive implementations, observe workings of OS

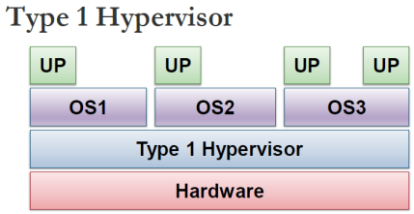
→ Software emulation of hardware, illusion of complete hardware(memory, CPU…) to level abv

- Normal OS can run on top of VMs

→ Managed by Hypervisor, or Virtual Machine Monitor (VMM)

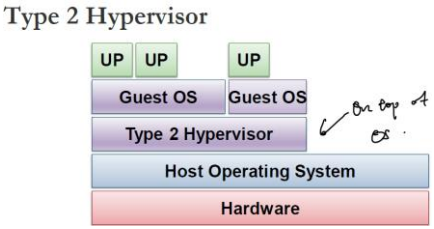
Type 1 Hypervisor

→ Lives directly above hardware



Type 2 Hypervisor

- Runs in host OS, guest OS runs on top of it (VirtualBox, VMWare)



Process Abstraction

Memory Context – Text, Data, Stack, Heap

→ On calling fork, ALL memory including text, data copied

Hardware Context – General Purpose Registers, Program Counter, page directory base reg, TLB

OS Context – PID, process state, priority, time quantum

Function Calls

Control flow

Setup parameters → transfer control to callee → setup local variable → store result (if applicable) → return to caller

Issues (Control flow): We need to jump to function body without losing context of current function → store PC of caller

Issues (Data storage): We need to pass parameters to the function, capture the return result and have local variable declarations. We need a region of memory that is dynamic →

Stack

→ Contains return address of caller

→ Parameters for the function

→ Local variable storage

→ From higher to lower memory region

Stack pointer – Points at top of stack region, 1 slot AFTER last variable stored in that stack (?)

Stack Frame setup

- Prepare to make function call

→ Caller: Pass parameters using registers/stack

→ Caller: Saves return PC on stack (doesn't matter callee or caller saved registers)

- Transfer control from caller to callee. In callee:

→ Saves registers used that is about to be used by callee. Save old FP, SP. (callee saved)

→ Allocate space for local variables of callee on stack. Adjust SP to point to new stack top

Stack Frame Teardown

- Callee restores saved registers, FP, SP (can be done in caller if caller saved)
- Caller continues execution
- Use frame pointer (points to fixed location), to access certain variables in stack conveniently
- **Register Spilling** – when GPRs are exhausted, use memory to hold GPR value temporarily. Spill registers pre-function – callee/caller saved

Heap

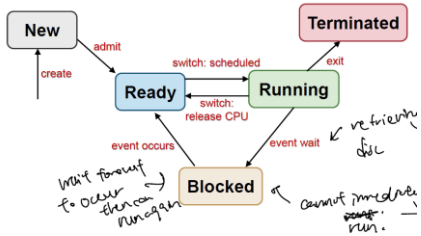
→ Runtime allocation, cannot place in Data or stack. Harder to manage as variables freed can cause 'holes' in memory

Process Management

→ Unique PID (can reuse, limits max processes)

→ PID 1 reserved for init process

5 States: New, ready , running, blocked, terminated

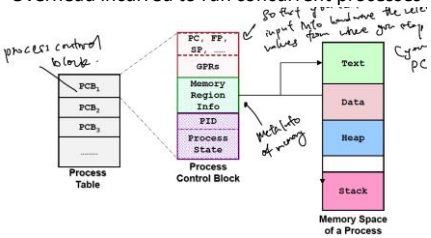


→ When running complex functions, take note the process might go from Ready → Running→ Ready→Running repeatedly

Process Control Block/Table

Stored as table representing all process in kernel

- Overhead incurred to run concurrent processes



- PC, FP, SP, GPRs, Memory region info etc stored in each entry of PCB. Only when process is swapped out, will hardware context (GPR) be updated.
- Memory context in PCB is not actual mem space used by process

System Calls

- API to OS, switches from user → kernel mode
- When we do the system call, library call places system call number in register, executes special instruction to switch from user to kernel mode (TRAP)
- System call handler saves CPU state, actual request is executed, cpu state is restored, library call switches from kernel to user mode again

Exception/Interrupts

- Exceptions are synchronous, occur due to program execution
- Interrupts are asynchronous, occurs independent of program execution

→ Suspends program execution, execute interrupt handler

When exception, interrupt occurs, control transfer to exception/interrupt handler routine automatically

PCB vs Process Table

A process control block (pcb) contains information about the process, I.e. registers, quantum, priority, etc.

The process table is an array of pcb's.

Process Abstraction in Unix

- PID (Integer)
- states: running, sleeping stopped, zombie
- Cumulative CPU time: Total amount of CPU time used so far
- ps for process information

fork()

Header File
```
#include <unistd.h>
```
Syntax
```
int fork( );
```

→ Creates new process
- Returns pid of newly created process (for parent), returns 0 for child
→ header file maybe different for other sys
- Behaviour: Creates a new process, child is a duplicate of current executable image
→ Memory is copied from parent (different text, data, heap, stack). Uses **copy on write**
→ Differs from parent only in process id (pid), parent (ppid), fork return value, memory locations
→ getppid() in child to get parent pid

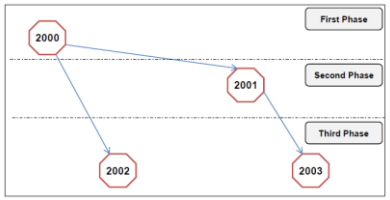
More about fork()

→ Since fork makes exact copy of parent process, i.e. new address space, new kernel process in process table (aka entry), copy kernel environment (for scheduling)
→ Child process context initialized: PID = process pid, PPID = parent id, zero CPU time
→ Copy memory region from parent, which is optimized by copy on write
→ Acquires shared resources (**open files**, curr working directory)
→ Init hardware context for child (copy registers etc from parent)
→ Child process then added to scheduler queue and is ready to run
- Copying memory is very expensive. Child doesn't access the whole memory range right away. Reads don't need to be copied. Can just use same address. Write needs to be different though, so **copy on write** is a possible optimization.

C code:
```
00 int main( ) {
01     //This is process P
02     if ( fork() == 0 ){
03         //This is process Q
04         if ( fork() == 0 ) {
05             //This is process R
06             ......
07             return 0;
08         }   `
09         <Point α>
10     }
11     <Point β>
12
13     return 0;
14 }
```

→ For question like this, notice that the child process inside might also run code at point B, so don't get tricked. If we put a wait at B, P waits for Q and Q waits for R.

Process Tree



→ In this process tree example, the pids in first phase/second phase all travel to third phase as well

Int main(int argc, char* argv[])
- argc: number of command line arguments, including program name itself
- argv: the arguments in string

exec()

Replace current executing process image with new one

Header File
```
#include <unistd.h>
```
Syntax
```
int execl( const char *path,
           const char *arg0,
           ...,
           const char *argN, NULL );
```

path – location of executable
null at end of arglist, remember to specify the executable again for arg0
→ returns errcode, if -1 then error occurred, else code wont execute from that point anymore if successful
→ fork() + exec() to get new process running for new program in unix

Master Process

→ init process, with pid = 1 (traditionally)
→ Created in kernel at bootup time, init is the root process
exit(int status) exit()
- status should be 0 for normal termination, non 0 for problematic execution. Does not return. Status returned to parent process
Process On Exit
→ When finished executing, most system resources used by process are released
→ E.g. file descriptor, each opened file in c has a file descriptor attached.
→ Some are not released: PID & status for parent-child synchronisation, process accounting info such as cpu time
→ If not cleaned up by parent, child remains in a zombie state. Process has run finished but still has entry on process table, resources are released
→ kill command has no effect on zombies
→ Return from main implicitly calls exit()
→ Clean up page tables and open files from open file table

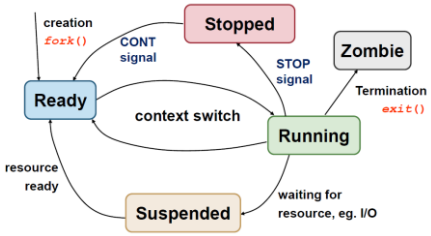
Wait(int *status) Wait()

Parent can wait for child process to terminate
→ Return pid of terminated child process
→ exit status of child stored in status pointer
→ Blocking wait, blocks until 1 child terminates
→ Remainder of child system resources cleaned up, **kills zombie process**
→ waitpid() for specific child, waited() to wait for any child process to change status – terminated child, stopped by signal, continued by signal
→ **Has no effect** if there are no children at all

Zombie vs Orphan

Orphan: Parent process terminates before child process, **init process becomes pseudo parent of child process**, child termination sends signal to init, which uses wait() to reap child

Zombie: child process terminates before parent but parent did not call wait(). Only thing is it fills up process table, taking up resources



Process Scheduling

Concurrent processes – multiple process progressing in execution. May not mean that there are multiple cores involved, but scheduling can achieve this effect
→ Interleaving on instruction from both processes: time slicing
- Interleaving requires OS to perform context switching between A and B, **incurs overhead**
Scheduling
- Each process has different cpu or i/o time requirements and process behaviours.
- Different scheduling algos are useful for different environments (batch, interactive..)
→ **Batch processing**, no user interactive, no need to be responsive.
Interactive, has active user interacting with system, should be responsive and have low, consistent response time.
Real time processing, have deadline to meet, bounded by real time (robots)
Scheduling Criterias
Fairness: should get fair share of cpu per process or per user, should not be **starved**
Utilization: All parts of system should be utilized, don't waste resource idling

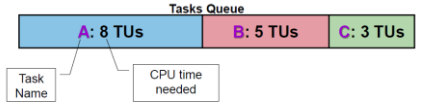
→ 2 types of scheduling policies. Defined by when the scheduling is triggered:
1. Non-preemptive (Cooperative), process stays scheduled (is in running state) until it blocks or gives up the CPU voluntarily
2. Preemptive scheduling, process is given fixed time quota to run, at end of quota, process is suspended, another process is picked to run

High Level Step by Step

1. Scheduler triggered (OS takes over)
2. (if needed) context of current running process saved and placed on block/ready queue. The resource might be given up by process itself or OS forces it preemptively
3. Pick suitable process based on queue to run scheduling algo
4. Set up Context for P. Run P

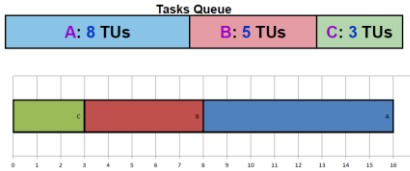
Batch Processing Scheduling Algorithms
- Batch: No user interaction, non-preemptive scheduling predominant
Criterias:
1. **Turnaround time** – **total time taken** = Finish time – arrival time.
Waiting time: time spent waiting for CPU
2. **Throughput**: Number of tasks finished per unit time, i.e. rate of task completion
3. **CPU/System utilization –** percentage of time when CPU is working on a task

FCFS (First come first serve)
- Tasks stored in a FIFO queue based on arr time
→ Task runs until done or blocked, blocked task removed from FIFO queue, when ready, placed at back of queue again
+ No starvation guaranteed, number of tasks in front of X can only decrease, so eventually will get its turn
- Can have very high avg waiting time.



Waiting time = 0 + 8 + (8 + 5) / 3 = 7
- Convoy effect: Consider first task like A which is CPU bound, followed by other I/O bound resources. All processes wait very long for A. I/O is idling. A switches to I/O. After all processes run on CPU, they are now blocked by A at I/O again.

SJF (Shortest Job First)
- Select task with smallest **total** CPU time
→ Need to know total CPU time of task in advance, need to guess if info not available
+ Minimizes average waiting time
- Starvation possible since biased towards short jobs, long jobs may never get a chance



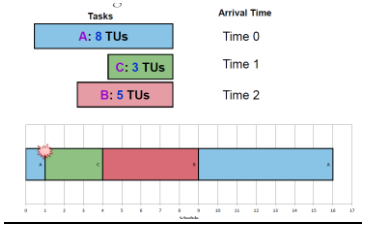
→ With a task queue of A,B,C, C is prioritized since it has shortest CPU time
→ Average waiting time: 0 + 3 + (3 + 5) / 3 = 3.66
→ SJF guarantees smallest avg waiting time
→ Can be preemptive or non preemptive

Predicting CPU Time
- Task goes through several phases of CPU Activity**: possible to guess the future CPU time** requirement by previous CPU-Bound phases
Exponential Average:

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

→ Taking weighted average of actual run time of prev task and predicted run time of prev task

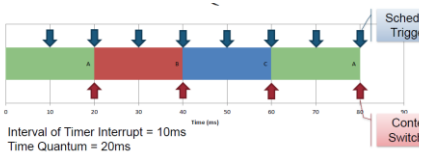
Shortest Remaining Time (SRT)
- Preemptive scheduling based on the remaining time of a task, select job with shortest remaining or expected time
→ New job with shorter remaining time can pre-empt currently running job. Good for short jobs that even arrive late, can starve long jobs



→ Task A was running. C comes in, preemptively Schedules C instead. Then we keep prioritizing tasks with SRT, A comes in only at the end

Scheduling for Interactive Systems
Criterias:
1. Response time: Time between req and res of system must be low, time from arrival to first allocation of CPU
2. Predictability: Less variation in res time == more predictable
→ Preemptive scheduling algo used to ensure good response time, scheduler needs to run periodically

Timer Interrupt, Interval of Timer Interrupt (ITI)
→ Allow scheduler to take over CPU periodically
→ Timer interrupt: Interrupt that goes off periodically based on hardware clock, cannot be intercepted by other program, invokes scheduler
Interval of Timer Interrupt:
- OS scheduler invoked on every timer interrupt, ranges typically from 1ms to 10ms
Time Quantum
- Execution duration given to a process
- Can be constant or variable among processes
- **MUST** be multiples of ITI, commonly ranges from 5ms to 100ms.
- Does not always change a process at every interrupt but it's the MAX time a process can run before it must be interrupted
- You can make a syscall to OS to trigger scheduler, don't have to just wait for ITI

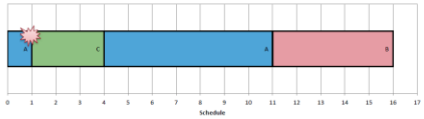


Round Robin (RR)
- Tasks stored in a FIFO queue, pick first task from queue to run until:
1. Fixed time slice (quantum) elapsed;
2. Task gives up CPU voluntarily;
3. Task blocks
- If time quanta elapsed, task is placed at end of queue to wait for another turn.
- If blocked, task placed in other queue to wait for requested resource. Placed at end of queue again when it is ready
- Process that comes in to first in line of queue when another process is running will have to wait till end of quanta to run

→ Basically premptive version of FCFS.
→ Response time guarantee: Given n tasks and a quantum q, time before task gets CPU is bounded by (n – 1)q (since there are n-1 tasks before it and they run max q time) i.e. no starvtn
→ Timer interrupt needed for scheduler to check on quantum expiry
→ Big quantum vs small quantum: Big has better CPU utilisation but longer waiting time, small has bigger overhead (worse cpu utilization) but shorter waiting time

Priority Scheduling
- Assign priority value to all tasks, select task with highest priority, as some processes are more important than others
- Preemptive and non-preemptive versions.
→ Higher priority process can pre-empt running process with lower priority
→ Non-pre-emptive: Late coming high priority process waits for next round of scheduling
- In this case, preemptive means that we cut in before quanta, non preemptive waits for quanta / task to finish (depend on scheduling reqs)

Tasks	Arrival Time	Priority (1=highes
A: 8 TUs	Time 0	3
C: 3 TUs	Time 1	1
B: 5 TUs	Time 1	5



→ C with highest priority cuts in on A, A has higher priority than B so it runs first

- For priority scheduling, low priority processes can starve. High priority processes hogs the CPU, situation is worse in preemptive variant
- Generally hard to guarantee or control exact amt of cpu time given to process using priority
→ Possible solution: Decrease priority of currently running process after every time quantum? Eventually, it will drop below next highest priority
→ Or, give current running process a time quantum. It will be considered in next round of scheduling

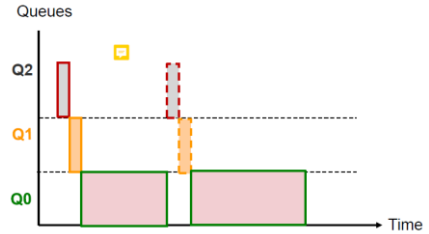
Priority Inversion
- Consider a scenario where Priority: {A = 1, B=3, C= 5} (1 is highest)
→ Task C locks a resource, task B preempts C before unlocking resource, A arrives and need resource but it is locked. Task B continues execution even though A has higher priority, known as priority inversion, where lower priority task pre-empts higher priority task

Multi-Level Feedback Queue (MLFQ)
- Motivation: How do we schedule without perfect knowledge? Since most algos need certain info (process behaviour, running time)

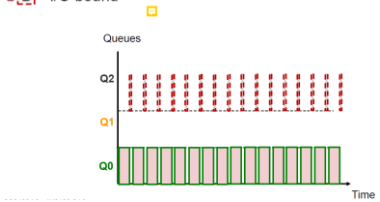
+ MLFQ is adaptive, 'learns' process behavour
+ It minimizes both response time for I/O bound processed and turnaround time for CPU bound processes
→ Note: **CPU and I/O have different schedulers, with different queues, if they both implement MLFQ, if the priority of a task in the CPU queue decreases, then when it is transferred over to the I/O queue, the priority is tracked separately for I/O (so the priority might be high again)**

Rules of MLFQ
1. If Priority(A) > Priority(B) → A runs.
2. If Priority(A) == Priority(B) → A & B runs in RR

Priority Setting:
1. New job has highest priority
2. If job fully utilizes its time quantum, its priority is reduced.
3. If a job gives up or blocks before finishing its time quantum, priority is retained.



→ Single job can go through 3 different queues, new job (dotted line) appears halfway, runs finish, task in Q0 takes over again
A = CPU bound (already in the system for quite some time)
B = I/O bound

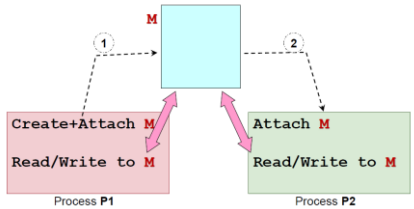


→ B is allowed to continually use the CPU resource before switching out, since it didn't use up quantum → MLFQ 'learns', i.e. adaptive
- MLFQ doesn't work well for long running tasks, they get pushed to low priority and might starve
→ Game system: Give up CPU before TQ

Lottery Scheduling
- General Idea: give out lottery tickets to processes for various system resources (cpu time, i/o devices etc)
- Lottery ticket chosen randomly among eligible tickets, winner granted the resource
- In the long run, process holding X% of tickets win X% of the lottery held, use resources X% of the time

+ Responsive: Newly created process can participate in the next lottery
+ Provides good level of control. Process can be given Y lottery tickets which can be redistributed to child processes
+ Important processes can be given more tickets, controlling proportion of usage
+ Each resource can have own set of tickets, for diffrnt proportion of usage per resource per task
+ Easy to implement

IPC (Inter-Process Communication)
→ Hard for cooperating processes to share information, as memory space is independent
→ Shared memory and Message passing (common IPC mechanisms)
→ Pipe and Signal (Unix-specific IPC)
Shared Memory
1. General Idea: Process P_1 creates a shared memory region M
2. P_1 and P_2 attach M to its own memory space
3. P_1 and P_2 can now communicate using M
→ M behaves very similar to normal mem region
→ **Any writes to the region are visible to other processes**
→ Applicable to more than 2 processes sharing same memory region
→ OS is only involved when processes create/attach to memory region (Step 1 and 2)



Pros and Cons of Shared Memory
+ Efficient: Only initial steps involves OS
+ Ease of use: Shm behaves the same as normal memory space, i.e. info or any type/size can be written easily
- Synchronization: Shared resrc -> **Need to sync access. Need to read value from memory, modify it, save value back into memory. Not atomic process**
- Harder to implement

How to use in *nix
- Basic steps of usage:
1. Create/locate a shared memory region **M**
2. Attach **M** to process memory space
3. Read from/write to **M**
   - Values written visible to all process that share **M**
4. Detach **M** from memory space after use
5. Destroy **M**
   - Only one process need to do this
   - Can only destroy if **M** is not attached to any process

```
#include <stdio.h>
#include <stdlib.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);
    }
```

The master program create the shared memory region and wait for the "slave" program to produce values before proceeding.

Step 1. Create Shared Memory region.

Step 2. Attach Shared Memory region.

```
    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;
}
```

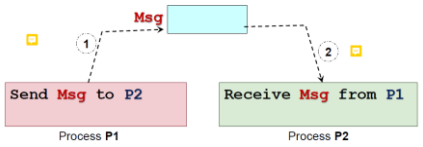
The first element in the shared memory region is used as "control" value in this example (0: values not ready, 1: values ready).

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

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

→ In above program, we create and attach to shm using shmget and shmat, then use first value (shm[0]) as control value, wait for elements to be ready from another process, detach and destroy shm using shmdt and shmctl.
→ Slave program basically just attaches to shm, write values to indices 1 to 3, indicate shm[0] as 1, the detach from shm

Message Passing
- General Idea: Process P_1 prepares a message M and sends it to P_2
- P_2 receives M
→ Message sending/receiving usually provided as syscalls
→ Other properties: Naming, to identify other party in communication; Synchronization is required, just like shared memory
→ Msg is stored in kernel memory space, so every send/rcv op must go through OS (ie syscall)
→ **Direct Comm / Indirect Comm**



Direct Communication

- Sender/Recver of message explicitly names other party (In unix, implementation is called unix domain socket)

→ API: Send(P_2 , Msg) [Send to P_2]

→ Receive(P_1, Msg) [Recv Msg from P_1]

→ One link per pair of communicating processes

→ Must know identity of other party

Indirect Communication

→ Message sent to/received from message storage: Known as mailbox/port (Unix implementation is called message queue)

→ API: Send(MB, Msg): Send msg to mailbox MB

→ Receive(MB, Msg): Receive msg from MB

→ 1 mailbox MB can be shared among a number of processes

→ Name of mailbox stored in file system, actual mailbox stored in kernel

Synchronization Behaviours of Mailboxes

→ Blocking Primitive (synchronous): Receive(): Receiver is blocked until msg arrives

→ Non-blocking Primitive (asynchronous): Receive(): Receiver either receives message if available or some indication that message is not ready yet.

→ Send can also block if buffer is full!

Pros and Cons of Message Passing

+ Portable: Can easily implement on different processing environments (dist. systems, wide area network…)

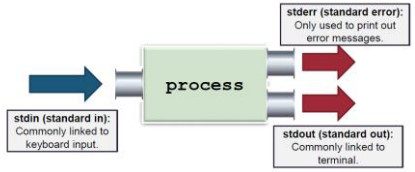
+ Easier synchronization: Sender/Recver implicitly synchronized when synchronous primitive used

- Inefficient: Requires OS intervention, extra copying of messages

Unix Pipes

→ General Idea: Communication channel created with read end and write end (2 ends)

→ 3 default communication channels in Unix: stdin(usually from keyboard), stderr(to print err msges), stdout(commonly linked to terminal)



Piping: "|" symbol to link input/output channels of one process to another; A|B means the stdout of A is linked to stdin of B (as if coming from keyboard)

→ Pipe can be shared across 2 processes, in a producer-consumer relationship

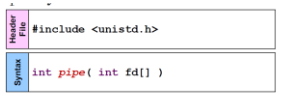
- P produces (writes) n bytes
- Q consumers (reads) m bytes

→ Behaviour: FIFO, data accesses in order, acts like an anonymous file (to the recving process)

→ Can be unidirectional (half-duplex) or bidirectional(full-duplex); but unidirectional are more common

→ Unix pipes semantics: Pipe functions as a **circular bounded byte buffer** with **implicit synchronisation:** Writer wait when buffer is full, Readers wait when buffer is empty

→ Other variants: Can have multiple readers/writers, compared to 1 reader/writer in normal pipe



- Returns:
  - 0 to indicate success; !0 for errors
  - An array of file descriptors is returned:
    - fd[0] == reading end
    - fd[1] == writing end


```
#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]);
   }
}
```

→ Remember we can attach/change std communication channels to one of the pipes, redirecting input/output from 1 prog to another

→ dup/dup2

- dup2(fileid, STDOUT_FILENO) means you take the STDOUT_FILENO and point to fileid; i.e. normally program outputs to stdout, but stdout is pointing to fileid so go to fileid
- dup2(fid, STDIN_FILENO) means you take the STDIN_FILENO and point to fid; normally program input from stdin, but stdin is pointing to fid so come from fid

Unix Signal

→ Form of inter-process communication, an asynchronous notification of an event.

→ Can be sent to process/thread

→ Recipient of signal handles signal using either default handlers OR user defined handlers

→ Common signals in Unix: Kill, interrupt, stop, continue, memory err, arithmetic err..

→ SIGKILL, SIGINT, SIGSTOP, SIGTSTP…

→ signal(SIGINT, handleInterrupt) to attach your own handler to that signal. Must reattach everytime after use

→ Cannot use user-defined handler for SIGKIL (kill -9)

Threads

→ Process creation is expensive: need to duplicate memory space and most of process context

→ Context switch requires saving/restoring of process information – computationally expensive

→ Hard for independent processes to communicate with each other; requires IPC

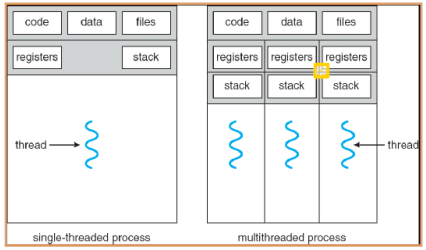
→ Single process can have multiple threads (aka multithreaded process)

→ Threads in same process shares:

1. Memory Context: Text, Data, Heap (diff stack)
2. OS Context: PID, file resources etc

→ Other hand, threads differ in:

1. Identification (thread id)
2. Registers (GPRs and special)
3. Stack



→ Between multiple threads, there is still a need to do context switching, just like in processes. But less stuff to switch

Process Context Switch VS Thread Switch

Process Context Switch involves:

- OS Context, hardware context, memory context

Thread Switch within a process involves:

- Hardware Context (registers, stack) only

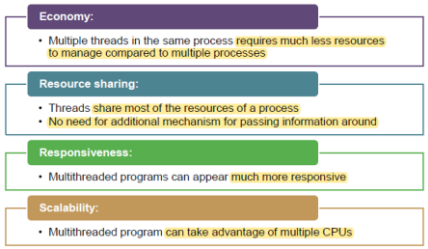
→ Much more **lightweight**

Process vs Threads

→ Multiprocess offers **isolation/protection:** Another process unresponsiveness/being blocked does not affect other processes, unliked in threads – good in modern browsers

→ Thread model **offers lower overhead** and is more suitable for less memory intensive programs or responsiveness required – good for games

Benefits of Threads



→ Easy to pass info around in threads

→ Can take advantage of multiple CPUs (usually 1 process cannot use multiple CPUs)

Problems with Threads

- System call concurrency

→ Since there are parallel executions of multiple threads, there might be a chance of parallel syscalls, which has to be handled to guarantee correctness and determine correct behaviour

- How are thread behaviours defined

→ For e.g., fork() duplicates processes, then how should we handle the threads? This is usually system dependent, but in unix, you only fork 1 thread, not all n threads

→ If a single thread executes exit(), the entire process will exit

→ If a single thread calls exec(), what happens to other threads? System dependent. But in unix, all threads disappear except the 1 running binary

Thread Models

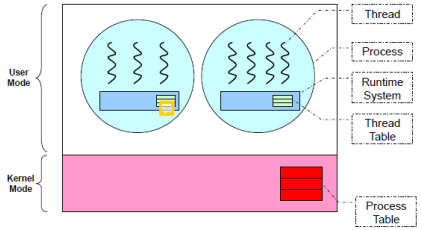
→ User threads and Kernel Threads

→ User Threads: Implemented as user library. Runtime system (in the process) will handle the thread related operation, **kernel is not aware of the threads in the process**

→ Kernel Threads: Implemented in the OS.

- **Thread operations are handled as syscalls.**
- Thread-level scheduling is possible – Kernel schedules by threads instead of by process (or is it schedule by both thread and process? Check)
- Synchronisation, context switching, creation all involve kernel for kernel threads
- Kernel may create its own kernel-level threads to perform tasks such as those above

User Thread Illustration:



→ Thread table managed by user library, in the process itself. Kernel manages the process table

User Threads Pros and Cons

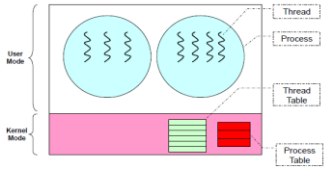
+ Can have multithreaded program on any OS.

+ Thread operations are library calls, not syscalls

+ More configurable / flexible (e.g. customize own thread scheduling policy)

- OS not aware of threads, scheduling performed at process level. **If one thread blocked → Process blocked → all threads blocked**
- Cannot exploit multiple CPUs

Kernel Thread Illustration:



→ Kernel holds both the process table and thread table

Kernel Threads Pros and Cons

+ Kernel schedules at thread level. More than 1 thread in same process can run simultaneously on multiple CPUs

- Thread operations now a syscall, which is slower and more resource intensive
- Generally less flexible. The thread policy defined is used by any process → If its is implemented with many features, more expensive and overkill for simple programs, but if implemented with too little features, then not flexible enough for some programs

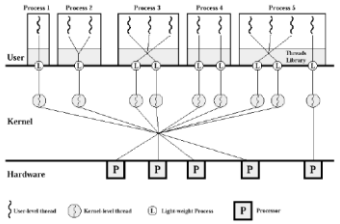
Hybrid Thread Model

→ Have both kernel and user threads. OS still only schedules on kernel threads, but user thread can bind to a kernel thread!

→ Offers great flexibility: Different kernel threads can run on different CPUs, and since within a single kernel thread we can run more than 1 user thread, we can perform efficient context switching between the user threads, since they are not syscalls

→ Can limit the concurrency of any process/user (?)

Hybrid Model Example: **Solaris**



Threads on Modern Processors

→ Threads started off as a software mechanism. Started off from user library (user threads) to OS aware (kernel threads)

→ Now, there are hardware support as well on threads, there are dedicated sets of GPRs and special regs to allow threads to run natively and in parallel on same core → simultaneous multithreading (e.g. 2 physical cores → 4 virtual cores = 4 threads) i.e. can run at same time

POSIX Threads - pthread

→ Only API behaviour defined, so can be implemented as user/kernel thread

Basics of pthread

- Header File:
  - `#include <pthread.h>`
- Compilation (flag is system dependent):
  - `gcc XXXX.c -lpthread`
- Useful datatypes:
  - `pthread_t` : Data type to represent a thread id (TID)
  - `pthread_attr`: Data type to represents attributes of a thread

pthread Creation Syntax

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

- Returns (0 = success; !0 = errors)

→ tidCreated: Thread Id for created thread. Put your own ptr in

→ threadAttributes: Control behaviour of new thread (Can put this as NULL in simple use)

→ startRoutine: Function pointer. Thread executes this function

→ argForStartRoutine: startRoutine's arguments

pthread Termination

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

- Parameters:
  - `exitValue`: Value to be returned to whoever synchronize with this thread (more later)

→ Exit value is return value of thread, which is returned to whoever synchronizes with this thread (aka use join on it)

→ If pthread_exit() not used, a pthread will terminate automatically at the end of startRoutine. If we do not explicitly define a return value in the startRoutine function, then the exitValue will not be well defined. If we don't need an exitValue (void function) can put NULL

```
//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;
}
```

→ spawning multiple ptheads using pthread_create within a single process can cause **race conditions**. **If all threads are reassigning a global variable, the end value of the global variable is undeterministic**

pthread_join – Simple synchronization

- Wait for termination of another pthread

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

- To wait for the termination of another ***pthread***:
- Returns(0 = success; !0 = errors)

→ threadID: TID of pthread to wait for

→ status: Exit value returned by target pthread

Other functions of pthread

- Yielding (giving up CPU voluntarily) by doing pthread_yield
- More: Advanced synchronization, scheduling policies, binding to kernel threads etc

Synchronization

Motivations: Synchronization problems occur when 2 or more processes execute concurrently in interleaving fashion and sharing a modifiable resource

→ Execution of a single sequential process is deterministic. However, execution of concurrent processes may be non-deterministic; execution outcome depends on the order in which the shared resource is accessed/modified

→ Known as race conditions

→ Simplest example – modifying a global variable requires 3 steps: Loading variable to register, modifying, storing back in register. If another process is doing the same thing and the steps are interleaved, the final result in memory will not be accurate

Race Condition: **Bad behavior**

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

→ Solution: Designate code segment as critical section, only 1 process can execute in critical section

Properties of Correct Critical Section Implmtn

1. Mutual Exclusion – If process P_i is executing in critical section, all other processes are prevented from entering the critical section

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

3. Bounded Wait: After process P_i request to enter critical section, there exists an upper bound on the number of times other processes can enter the critical section before P_i.

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

Symptoms of Incorrect Synchronization

- Deadlock → All processes blocked, no progress
- Livelock → Usually related to deadlock avoidance mechanism, processes keep changing state to avoid deadlock and make no progress
- Starvation → Some processes never get to make processes because perpetually denied necessary resources (no bounded wait)

Test And Set: Assembly Level Impl of CS

Behaviour: Load current content at MemoryLocation into Register, Stores a 1 into MemoryLocation

→ Behaviour above performed as atomic op

```
TestAndSet Register, MemoryLocation
```

TestAndSet() takes a memory address M:
- Returns the current content at M
- Set content of M to 1

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

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

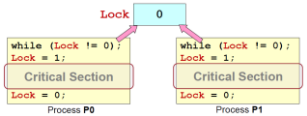
→ Works, but employs busy waiting, wasteful use of processing power

→ Other variants: Compare & Exchange, Atomic Swap, Load Link/Store Conditional

High Level Language Implementations of CS

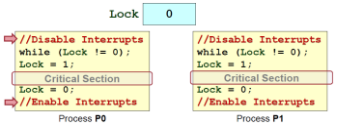
Naïve attempt:

→ Use global variable "lock", where whoever wants to enter CS set the lock = 1, lock = 0 to exit



→ Interleaving can still occur, mutual exclusion not satisfied

→ What if we disable interrupts?



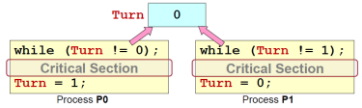
→ Works, but don't do it because:

- This will not work in a multi-core/multi-processor environment since another proc may enter the critical section while running on a different core.
- User code may not have the privileges needed to disable timer interrupts
- Disabling timer interrupts means that many scheduling algorithms will not w properly
- Disabling non-timer interrupts means that high-priority interrupts that may not e share any data with the critical section may be missed.
- Many important wakeup signals are provided by interrupt service routines and th would be missed by the running process. A process can easily block on a semaph and stay blocked indefinitely, because there is nobody to send a wakeup signal.
- if a program disables interrupts and hangs, the entire system will no longer work si it cannot switch tasks and perform anything else.

→ Busy waiting, requires permission to disable/enable interrupts

Attempt 2

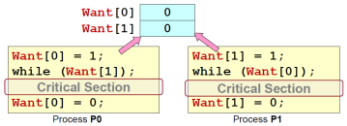
→ Use a global variable "turn" to determine who's turn is it



→ Violates independence property: If P0 never enters CS, P1 will **starve**

Attempt 3

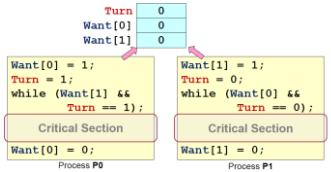
→ Have 2 global variables, representing who wants to go into CS



→If both processes indicate Want[i] = 1, then **deadlock** at busy wait portion

Peterson's Algorithm

→ Have Turn and Want variables. Represents who's turn and who wants to enter CS



→ Works, assume writing to Turn is an atomic operation

Disadvantages of Peterson's Algorithm

- Busy Waiting: Waiting process checks while loop condition instead of going to sleep/blocked
- Low level Implementation: More error prone, more complicated mutual exclusion mechanism compared to HLL programming construct
- Not general: Doesn't really extend to more than 2 processes / arbitrary number of processes

Semaphores

→ Generalized synchronisation mechanism, behaviour is specified but not impl

→ Provides a way to block a number of processes (aka sleeping process)

→ Provides a way to unblock/wakeup 1 or more sleeping process

→ General Semaphores (S>=0 (e.g. S = 0,1,2..)) or Binary Semaphores (S = 0 or S = 1)

→ Usually, binary semaphores are sufficient, general provided for convenience

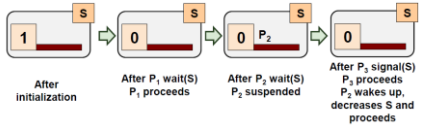
Semaphore Wait() and Signal()

→ Semaphore contains an integer value, can be initialized to any non-negative values initially

→ Wait > 0 = you can go in

`Wait( S )`	`Signal( S )`
If S <= 0, blocks (go to sleep)	Increments S
Decrement S	Wakes up one sleeping process if any
Also known as `P()` or `Down()`	This operation **never** blocks
	Also known as `V()` or `Up()`

Semaphore Visualisation



→ Semaphore has a protected integer, with a list of waiting processes to be kept tracked of

Semaphore Properties

- Given:
  - $S_{Initial} \geq 0$

Then, the following **invariant** must be true:

$$S_{current} = S_{Initial} + \#signal(S) - \#wait(S)$$

Semaphore CS

```
Wait( S );
Critical Section
Signal( S );
```

Usage of semaphore as CS is known as mutex (mut. excl.)

Semaphore Informal Correct CS Proof

Mutex Fulfilled:

- N_{CS} = Num of processes in critical section
  = Num of processes 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 $S_{current} \geq 0$ → $N_{CS} \leq 1$

Prevent Deadlock:

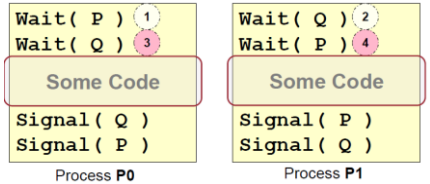
→ Assuming Sinitial = 1,

- Deadlock means **all** processes stuck at `wait( S )`
  → $S_{current} = 0$ and $N_{CS} = 0$
- But $S_{current} + N_{CS} = 1$
- →← (contradiction)

Prevent Starvation:

- Suppose P1 is blocked at `wait( S )`
- P2 is in CS, exits CS with `signal( S )`
  - If no other process sleeping, P1 wakes up
  - If there are other process, P1 eventually wakes up (assuming fair scheduling)

→ **Incorrect use of semaphore can still lead to deadlock (interleaving)**



→ Assume initial P and Q = 1, step 1 (P = 0), step 2 (Q = 0), step 3 blocks, step 4 blocks. Deadlock

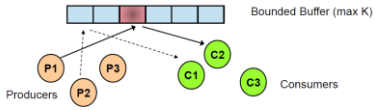
Synchronization Problems

Producer Consumer

- Processes share a bounded buffer size K

→ Producers produce items to insert in buffer, only when buffer not full (i.e. < K items)

→ Consumers remove items from buffer, only when buffer not empty (i.e. > 0)




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

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

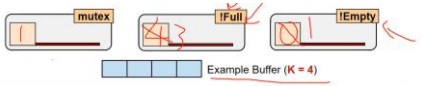
→ Initial Values: count = in = out = 0, mutex = S(1), canProduce = true, canConsume = false

→ Produce/Consume outside of mutex because that might be expensive operation. Do at the same time

→ Correctly solves BUT uses busy-waiting

Producer Consumer: Blocking Version

→ Puts the other waiting processes to sleep




```
while (TRUE) {
    Produce Item;
    wait( notFull );
    wait( mutex );
    buffer[in] = item;
    in = (in+1) % K;
    count++;
    signal( mutex );
    signal( notEmpty );
}
```
Producer

```
while (TRUE) {
    wait( notEmpty );
    wait( mutex );
    item = buffer[out];
    out = (out+1) % K;
    count--;
    signal( mutex );
    signal( notFull );
    Consume Item;
}
```
Consumer

- `wait( notFull )` : Forces producers to go to sleep
- `wait( notEmpty )` : Forces consumers to go to sleep
- `signal( notFull )` : 1 consumer wakes up 1 producer
- `signal( notEmpty )` : 1 producer wakes up 1 consumer

→ No busy waiting

→ notFull – number of space in buffer

→ notEmpty – number of items in buffer

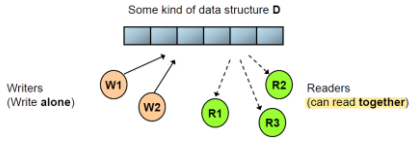
Reader Writer Problem

- Processes share a data structure D

→Reader: Retrieves Info from D

→ Writer: Modifies Info in D

→ Writer requires exclusive access to D, reader can access with other readers




```
while (TRUE) {
    wait( roomEmpty );
    Modifies data
    signal( roomEmpty );
}
```
Writer Process

```
while (TRUE) {
    wait( mutex );
    nReader++;
    if (nReader == 1)
        wait( roomEmpty );
    signal( mutex );
    Reads data
    wait( mutex );
    nReader--;
    if (nReader == 0)
        signal( roomEmpty );
    signal( mutex );
}
```
Reader Process

- Initial Values:
  - roomEmpty = S(1)
  - mutex = S(1)
  - nReader = 0

→ revDoor = S(1) Initially as well

→ Only one reader can enter – use binary semaphore

→ Multiple Readers can enter – we only update binary semaphore if he is first reader/last reader in the room

→ We have a global variable nReaders – wrap in mutex so the value is consistent

→ One issue: Writer might get starved if readers keep coming in

→ Solution? Have one more semaphore, a revolving door, where the writer who wants to enter will 'block' the door, and only enter when everyone else leaves the room

```
Semaphore revDoor = 1;
Original code in black color, new code in red.
Writer
wait(revDoor); // blocks the "door"
wait(emptyRoom);
modifies data
signal(emptyRoom);
signal(revDoor);
```

```
Reader
wait(revDoor);
signal(revDoor); // immediately let another task pass through
                 // UNLESS the writer "blocks" the door.
wait(mutex);
nReader++;
if (nReader == 1)
    wait(roomEmpty);
signal(mutex);
Reads data
wait(mutex);
nReader--;
if (nReader == 0)
    signal(roomEmpty);
signal(mutex);
```

Dining Philosophers' Problem

- Five philosophers are seated around a table
  - There are five single chopstick placed between each pair of philosopher
  - When any philosopher wants to eat, he/she will have to acquire both chopsticks from his/her left and right
- Devise a **deadlock-free** and **starve-free** way to allow the philosopher to eat freely

Attempt 1

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

//For philosopher i
while (TRUE){

    Think( );
    //hungry, need food!
    takeChpStk( LEFT );
    takeChpStk( RIGHT );

    Eat( );

    putChpStk( LEFT );
    putChpStk( RIGHT );
}
```

- Can you figure out the problem?

→ If all the philosophers take the left chopstick at the same time, nobody can take right chopstick, deadlock

→ What if we make philosopher put down left chopstick if right chopstick cant be acquired? Livelock

Attempt 2

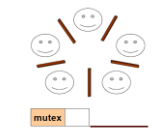
```
#define N 5
#define LEFT i
#define RIGHT ((i+1) % N)

//For philosopher i
while (TRUE){

    wait( mutex );

    takeChpStk( LEFT );
    takeChpStk( RIGHT );
    Eat( );
    putChpStk( LEFT );
    putChpStk( RIGHT );

    signal( mutex );
}
```



- Two questions:
  - Does it work?
  - Is it good?

→ Works but only 1 philosopher can eat at one time (because mutex), when we could have 2

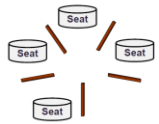
Limited Eater

→ Have at most 4 philosophers sit at the table

→ Impossible to deadlock

→ **Deadlock is impossible!**

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



- Initial Values:
  - seats = S(4)
  - chpStk = S(1)[5]

POSIX Semaphore

- Popular implementation of semaphore under Unix
- Header File:
  - `#include <semaphore.h>`

→ Compile with gcc something.c -lrt

→ Basic usage: Init semaphore, perform wait() or signal() on semaphore

Pthread Mutex / Conditional Variables

- Synchronization mechanisms for pthreads
- Mutex (`pthread_mutex`):
  - Binary semaphore (i.e., equivalent Semaphore(1)).
  - Lock: `pthread_mutex_lock()`
  - Unlock: `pthread_mutex_unlock()`
- Conditional Variables(`pthread_cond`):
  - Wait: `pthread_cond_wait()`
  - Signal: `pthread_cond_signal()`
  - Broadcast: `pthread_cond_broadcast()`

Tanenbaum Solution

→ 3 states for philosophers, state array of each philosophers' state

→ Key: You don't grab mutex for entire duration of eating, so someone else can eat as well

→ Another philo may signal a philo's state semaphore

→ First guy can take chopsticks and eat. If his neighbors want to eat, safeToEat fails, they will be blocked on their own wait. Then when guy finishes eating, he call safeToEat for them, which signals, completing takeChpSticks for both neighbors, then they will eat. So mainly, if someone is hungry but cannot eat because no chopsticks, they will be stuck on takeChpSticks

```
#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] );
}
```

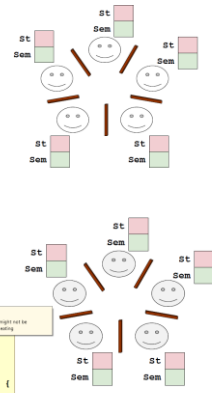
```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

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

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

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

    signal( mutex );
}
```



Memory Abstraction

RAM – Each byte has a unique index, known as a physical address.

→ Contiguous memory region with an interval of consecutive addresses

Transient Data

→ Valid only for a limited duration (e.g. during a function call), e.g. parameters, local variables

Persistent Data

→ Valid for the duration of program unless explicitly removed (e.g. global variable, dynamically allocated memory)

→ Both types can grow/shrink during execution

OS: Managing Memory

OS handles:

→ Allocation of memory space to new processes

→ Managing memory space for processes

→ Protecting memory space of processes from each other

→ Provides memory related system calls to processes

→ Manage memory space for internal use

Memory Abstraction

Without memory abstraction:

+ Memory access is straightforward (Address in program == physical addr, no conversion / mapping req, address fixed during compilation)
- Two processes cannot occupy same physical memory, conflicts happen if both assumed to start at 0, for example
- Hard to protect memory space

Fix Attempt: Address Relocation

→ Recalculate memory references when the process is loaded into memory (e.g. add offset of 8000 to all mem references in process B)
- Slow loading time (must add to entire program)
- Not easy to distinguish memory reference from normal integer constant

Fix Attempt 2: Base + Limit Registers

1. Use special register as base of all memory references, **Base Register**

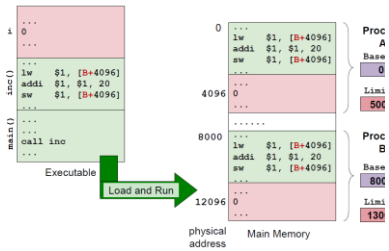
→ All memory references compiled as offset from this register at compile time

→ Base register initialized to starting address of process memory space at loading time

2. Add another special register to indicate range of memory space of current process: **Limit Register**

→ All memory accesses checked against this limit to protect memory space integrity

Base + Limit Register: Illustration



- Every memory access incurs an extra addition and comparison step: Actual = Base + Addr && Actual < Limit

+ Can be generalized into segmentation mechanism (both forked processes might think that they are using same address (spaces) but not so in hardware)

Logical Address

- Refers to how process views its address space

→ Embedding physical memory address in program is a bad idea

→ Have a mapping between logical and physical address

→ Each process has self-contained, independent logical memory space

Contiguous Memory Management

→ Process must be in memory during execution
- Stored-program counter, load-store memory execution model

→ Assume that each process occupies a contiguous memory region

→ The physical memory is large enough to contain one or more processes with complete memory space

Multitasking, Context Switching & Swapping

Support Multitasking by:
- Allow multiple processes in physical memory at the same time
- Switch from one process to another

When physical memory is full:

→ Free up memory by:
- Removing terminated process
- Swapping blocked process to secondary storage

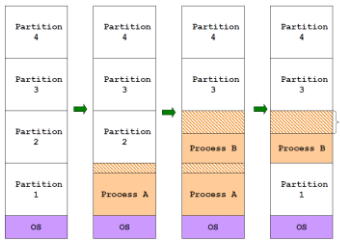
Memory Partitioning

- Memory Partition: Contiguous memory region allocated to a single process

→**Fixed-Size Partition**

- Physical memory split into fixed number of partitions, a process occupies one of the partition

Fixed Partitioning: Illustration



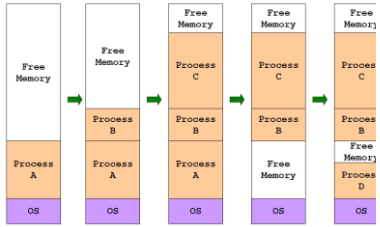
→ There is wasted memory space (**internal fragmentation**) within each memory block that is occupied by a process

+ Easy to manage, fast to allocate (every free partition is the same, no need to choose)
- Partition size need to be large enough to contain largest processes, smaller process will waste memory space (internal fragmentation)

→**Variable-size Partition**

- Partition is created based on actual size of process, OS keeps track of the occupied and free memory regions, performing splitting and merging when necessary

Dynamic Partitioning: Illustration



→ Can lead to **external fragmentation**

→ Free memory space known as a **hole,** known as **external fragmentation**. **Merge the holes by moving occupied partitions to create larger holes**

→ With multiple process creation / termination / swapping, tend to create large number of holes

+ Flexible and remove internal fragmentation
- Need to maintain more information in OS, takes more time to locate appropriate region

Dynamic Partitioning Allocation Algorithms

- Assume OS maintains a list of partition and holes

To allocate a partition of size N, we must first find a hole with size M > N, using:

→ First Fit, take first hole that is large enough

→ Best Fit, find smallest hole large enough

→ Worst Fit, Find the largest hole

- Then, split the hole into N and M-N

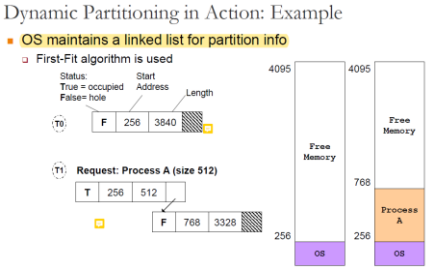
→ N will be new partition

→ M-N will be left over space (new hole)

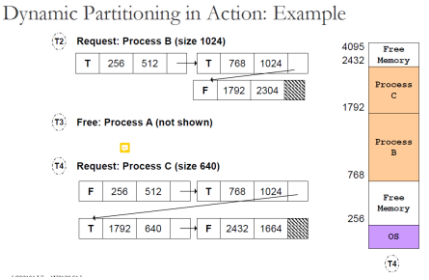
→ Next fit, following allocations after the first will not start from the beginning, but instead from the partition where the prev allocation was performed. Leads to more uniform distribution of hole sizes across free list, leading to faster allocation. But might increase the amt of external fragmentation

- When an occupied partition is freed, we merge with adjacent holes if possible
- Compaction can also be used; Move the occupied partition around to create consolidated holes
- **Very time consuming**

Dynamic Partitioning in Action: Example



Dynamic Partitioning in Action: Example



→Diagram: Using first fit algo, we first have partition info indicating 3840 length free space starting at 256. We split the free hole up to add the incoming A of size 512. Update the first node to T = occupied, change length to 512, add new free node of length 3840 – 512 = 3328

→ Update accordingly again when B comes in. When A is freed, indicate the first node to have F. We now have holes. Check if can merge front and back (not possible here)

Allocation: O(N)

Deallocation: We can store pointer to previously allocated nodes, so that's O(1)

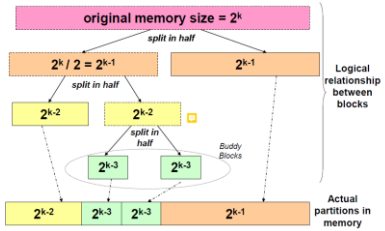
Merging: O(1)

Buddy System

→ Provides efficient partition splitting, locating good match of free partition, partition deallocating and coalescing

→ Split free block into half repeatedly to meet request (the 2 halfs form sibling / buddy blocks) When buddy blocks are both free, can be merged to form larger block. They must come from same block to be considered buddies!

Buddy Blocks Illustration



→ Look at actual partitions, 2^{k-2} in memory but its buddy has been split into 2^{k-3}, not in memory

Implementation

→ Keep an array A[0…K], where 2^k is largest block size that can be allocated

→ Each A[J] is a LL which keeps track of free blocks of size 2^J , indicated by starting address

→ Might have a 'smallest block size' that can be allocated in actual impl as well

→ If block is too small, its not cost effective to manage

- Buddy system results in both internal and external fragmentation.

Internal – If space you require is not power of 2, internal fragmentation

- External – If the buddies are not split properly, we ideally want a form of compaction/merging to move used spaces around. However, this is not possible in buddy system
- Merging not possible (we only merge with our buddy)

Buddy System Allocation Algorithm

To allocate block of size N:

1. Find smallest S, such that $2^S >= N$ (e.g. If N = 120, smallest S = 7 (1-based) because $2^7 = 128$

2. Access A[S] for free block

→ If exists, remove block from free block list, allocate block

→ Else, find smallest R from S+1 to K where A[R] has free block B

For (R-1 to S): Repeatedly split B until block is just big enough for N, now A[S…R-1] has a free block

3. go to step 2

Buddy System Deallocation Algorithm

1. To free block B:

Check in A[S], where 2^S == size of B

2. If buddy C of B exists (is also free), remove B and C from list, merge B and C to get larger block B'

2a. Go to step 1, taking B = B'

2b. If not free, Insert B to list in A[S]

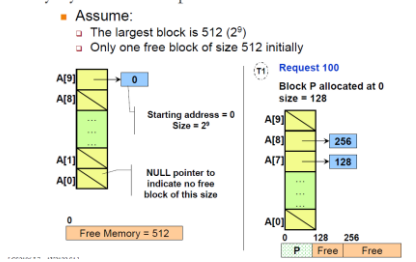
Identifying Buddies with Bits

→ Given block address A is 000, size = 4

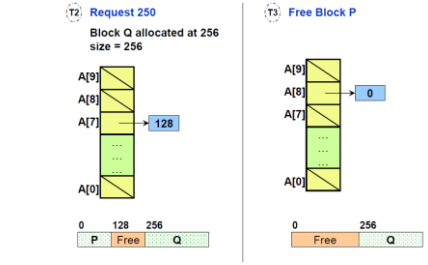
→ When we split into 2 blocks of half the size: B = 000, C = 010

→ 2 blocks B and C are buddies of size 2^S, if the S^{th} bit of B and C is a complement (in this case, the 2^{nd} bit is the same)
→ The leading bit up to the S^{th} bit of B and C are the same (both 0s in this case)

Buddy System: Example



Buddy System: Example (cont)



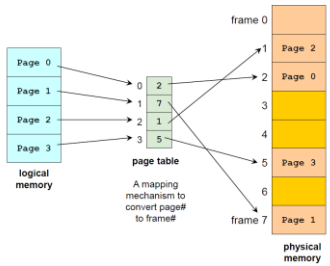
→ Initially, we have 512 B of free memory, only one block of size 512
→ Block P of size 100 requested, 128 given (starting from smallest S).
→ Block Q of size 250 requested, 256 given
→ Free block P, add starting address 0 to block 7, merge together, then now it is in block 8 with starting address 0
→ The numbers in the nodes are the starting addresses (0, 128, 256)

Disjoint Memory Schemes
- Process memory space can be in disjoint physical memory locations via paging
→ Physical Memory is split into regions of fixed size (Physical Frames)
→ Logical memory of process split into regions of same size (Logical Pages)
→ size(logicalPage) = size(physicalFrame)
→ Pages of process loaded into any available physical frame, logical memory space **contiguous** but occupied physical memory **disjoined**

→ Contiguous mem allocation, just need starting address and size of process as metadata

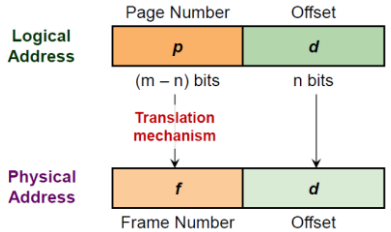
Paging – Need page table to translate logical page <-> physical frame

Paging: Illustration

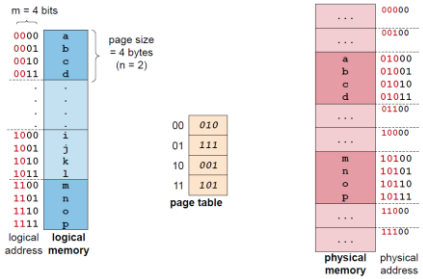


→ Program code uses logical memory address
→ Use physical memory address to access actual value.
→ Locate value in physical memory => Need **F, frame number** and **offset, displacement from start of frame**
→ **Phys Addr: F x size(physicalFrame) + offset**

Logical Address Translation
1. Keep frame size (or page size) as power-of-2
2. Physical frame size == logical page size



→ Page number contiguous, easy to translate by comparing the bit values
Formula:
- Given:
  - Page/Frame size of 2^n
  - m bits of logical address
- Logical Address LA:
  - p = Most significant m−n bits of LA
  - d = Remaining n bits of LA
- Use p to find frame number f
  - from mapping mechanism like page table
- Physical Address PA:
  - PA = f*2^n + d



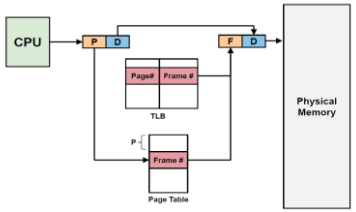
→ If page size = 4bytes, then we gonna use the first 2 bits in the address as the page number, check the mapping, replace page number with frame number, then match same offset for page to physical frame

→ Paging removes external fragmentation
→ No left-over physical memory region
→ All free frames can be used with no wastage

→ Still have internal fragmentation
→ Logical memory space may not be multiple of page size (100kb space on 128kb page size)

→ Clear separation of logical and physical address space
→ Separation of logical and physical addr space
→ allows great flexibility, easier to ask for more memory since not contiguous
→ simple to translate address, efficient

Implementing Paging Scheme
1. Software Solution
- OS stores page table alongside process information(e.g. PCB). Each process has own page table.
- Improved Understanding: Memory context of a process == page table
Issues: Require 2 memory accesses for every memory reference. 1st access to read the indexed page table entry to get frame number. 2nd access to access the actual memory item
2. Hardware Support
- Specialized on-chip component to support paging, known as TLB (Translation Look-Aside Buffer), acts as **cache** for a few page table entries, NOT PAGES.
- Logical address translation with TLB: Use page number to search TLB **associatively**
→ TLB-Hit: Frame number retrieved to generate physical addr (1 TLB access + 1 mem access)
→ TLB-Miss: **Access full page table**, retrieved frame number used to generate physical addr and update TLB (1 TLB + 2 mem access + 1 TLB update)



TLB: Impact on Mem Access Time
- Memory access time
= TLB hit + TLB miss
= 40% x (1ns + 50ns) + 60%(1ns + 50ns + 50ns)
= 81ns
- Note:
  - Overhead of filling in TLB entry and impact of cache ignored.

→ 1ns TLB access vs 50ns mem access
→ Mem access in TLB not uniformly distributed.
Locality principle: it is more likely to have repeated memory accesses to same (temporal locality) or different (spatial locality) parts of the same memory page in a time interval rather than uniformly spread accesses

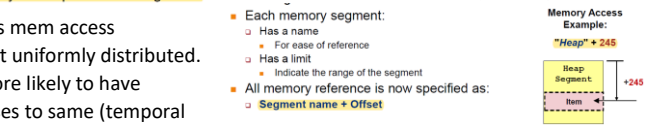
TLB and process switching
→ **TLB part of hardware context of a process**
→ Context Switch: TLB flushed, new process will not get incorrect translation
→ When process resumes running: 2 implementations, either TLB missed until filled again, or place some code pages initially to reduce TLB misses

Paging Scheme: Protection
Access-Right Bits and Valid Bit to provide memory protection between processes
Access Right Bits: Each pte includes several bits to indicate whether page is writable, readable, executable
→ **Always check against access right bits when performing memory access**

Not all processes utilize whole range of logical memory range (i.e. it is out of range for process)
- Valid bit:
  - Attached to each page table entry to indicate:
    - Whether the page is valid for the process to access
  - OS will set the valid bits when a process is running
  - Memory access is checked against this bit:
    - Out-of-range access will be caught by OS

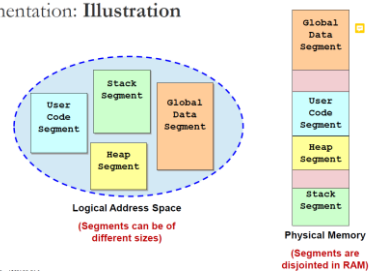
→ **Valid bit cannot catch out of bound memory that is within page boundary!!!!**
Page Scheme: Page Sharing
→ Observation: Several processes share the same physical memory frame.
→ Usage: Shared code page: Some code are being used by many processes, i.e. C stdlib, syscalls
→ **Copy on write**: Child and parent process share page until one tries to change value in it. (Child changes when any of them try to change vals)
OS checks that this is a copy-on-write page (additional information).
a. allocate new frame
b. copy current content over to (a)
c. Re-perform the memory store

Segmentation Scheme
- Separate regions into multiple memory segments. Logical memory space of process is collection of segments (i.e. heap, text, data, stack)
- Name + Limit to indicate range of segment



→ Heap + 245 to specify which that part of mem

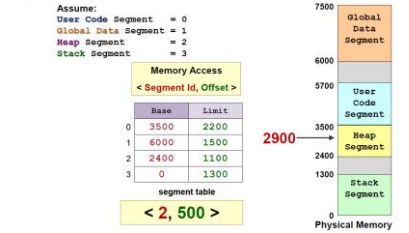
Segmentation: Illustration



→ **Memory is all contiguous in segmentation scheme**

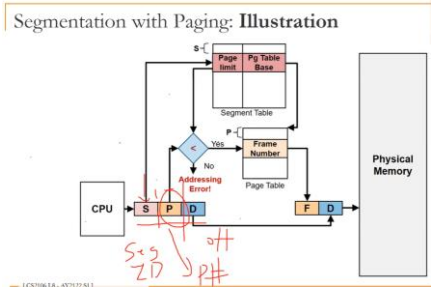
Segmentation: Logical Address Translation
- Each segment mapped to a contiguous physical memory region with **base address + limit**
- Use segment id to represent segment name
Logical Addr <SegID, Offset>:
- SegID looks up <Base, Limit> of segment in segment table
- **Phy Addr PA = Base + offset, Offset < Limit for valid access**

Logical Address Translation: Illustration



Segmentation: Summary
→ Pros: Each segment an independent contiguous memory space, can grow/shrink independently, can be protected / shared independently
→ We can find larger free piece of memory by relocating a heap/text… region, reassigning the <base, limit> values afterwards in base registers
→ Cons: Requires variable-size contiguous memory regions, can cause **external fragmentation**

Segmentation with Paging
- Each segment composed of several pages, i.e. each segment has a page table
- Segment grow/shrink by allocating/deallocating pages, adding/removing from page table



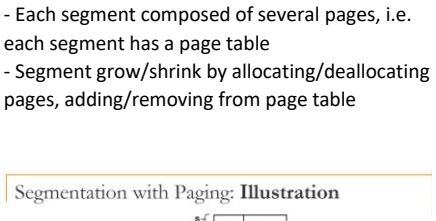
→ S – segment ID, p – page number, d – offset
→ page limit – highest page number of that segment, page table base – base address of the page table
Use P (page number) to index page table to get frame number, then can access physical memory
→ Limit in the Segment table now refers to the limit of **pages** in the corresponding page table i.e. **memory error that is in page boundary is not catchable**

Summary
→ Paging and segmentation solves different things. Paging allows us to load data into memory without it being in contiguous order, whereas segmentation allows each segment to grow independently and protects each segment independently

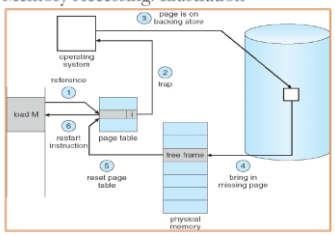
<u>Virtual Memory Management</u>
→ Logical Address space is split into pages, as discussed previously. Now, only some pages are in physical memory whereas some are stored on secondary storage (because secondary storage has much larger capacity, some programs cannot all fit into memory)

→ Still use page table to translate virtual/logical address to physical address
→ Memory resident for pages in physical mem
→ non-memory resident for pages in secondary storage
→ Add a is_memory_resident bit
→ OS triggers a page fault when CPU tries to access a non-memory resident page, OS brings non-memory resident to physical memory

Page Accessing Algorithm



→ OS **does not** delete page in secondary storage (lazy delete)
→ Update entry in page table, is resident bit = 1



→**Thrashing:** when memory access results in page fault most of the time, non-memory resident pages need to be loaded, keep accessing secondary storage
→ Why is thrashing unlikely to happen?
Temporal Locality and Spatial Locality

Temporal / Spatial Locality
→ Most time is spent on small region of code
→ Within a time period , accesses are only made to small part of data
Temporal: Memory address which is referenced is likely to be referenced again
→ cost of loading a page is **amortized**
Spatial: Memory address close to a referenced address is likely to be referenced again
→ Later accesses to nearby addresses will not cause a page fault

Programs can still behave badly due to poor design/malicious intent.

Virtual Memory Summary
→ Complete separate logical memory addresses from physical memory, amt of physical memory no longer restricts the size of the logical memory space!
→ More efficient use of physical memory: Page currently not needed is on secondary storage
→ Allow more processes to reside in memory: Improve CPU utilisation as there are more processes to choose to run (think scheduling)

On Demand Paging
→ Only when process requires a page, load it into physical memory, only loading the pages on-demand. **Reduces start up time**

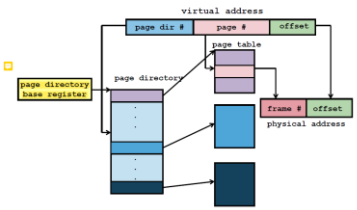
Page Table Structure
Issue: Modern computer systems have huge logical memory space → We can have many pages. Each page requires a page table entry (pte), resulting in large page tables
→ Incurs high overhead
→ Fragmented page table: Page table occupies several memory pages

Direct Paging
- Keep all entries in a single table
- With 2^p pages in logical memory space, p bits to specify one unique page
→ Each page contains physical frame number, valid/invalid, access right bits
- Example:
  - Virtual Address: 32 bits, Page Size = 4KiB
  - P = 32 – 12 = 20
  - Size of PTE = 2 bytes
  - Page Table Size = 2^{20} * 2 bytes = 2MiB (!)

2-Level Paging
- Process may not use entire memory space, wasteful to use full page table
Basic Idea:
- Split full page table into regions, only use a few regions. As memory usage grows, allocate more regions
→ Need directory to keep track of the regions

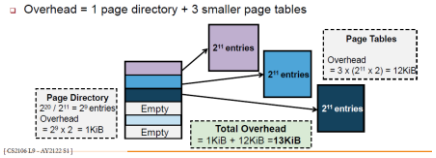
- Each page table has a page table number
→ **Each page table size = page size**
→ If page size is 4KB and 1 PTE size is 4 bytes, then each page table has 1024 pages, So if there were 2^{20} entries at first, there are now 2^{10} tables each with 1024 pages.
→ Always remember that each process has its own page table/page directory tables
- Page directory is needed to keep track of the smaller page tables
- Page directory contains 2^M indices to locate each of the smaller page tables

2-level Paging: Illustration



→ The page directory base register is updated as part of the **hardware context**

2-Level Paging: Advantages
→ Can have empty entries in page directory, so don't need to be allocated



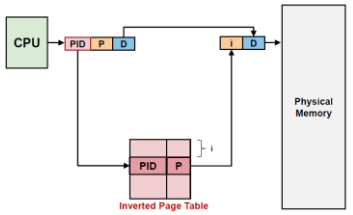
→ Page directory will still have overhead of all entries, but page table will only have overhead of the initialized tables.
→ 2^9 x 2 because each entry takes 2 bytes

Inverted Page Table
Observation: With M processes in memory, there are M independent page tables.
→ Difficult to find out which frames are occupied, and by which process
→ Only N physical frames can be occupied, which means that out of all entries in the M page tables, only N are valid
→ Waste in overhead: N << Overhead of M page tables

Idea:
- Keep single mapping of physical frame to <pid, page#>
→ pid + page# to uniquely identify page of a process, page# can repeat for different pids
→ Entries are ordered by frame number → Need to search the entire table to lookup a certain page X and which frame it belongs to

Advantage:
- One table for all processes: huge savings
- Easier and faster frame management
Disadvantage:
- Slow translation from page number to frame number (linear search, O(1), but bounded by number of physical frames (which can be q a lot))

Inverted Table: Illustration



Page Replacement Algorithms
→ When page fault occurs when accessed page is not in memory: We need to evict another memory page. We need to check if the page evicted is clean (not modified, no need to write back to storage) or dirty (modified, write back to storage)
→ **Only page number** is important because we are evicting the whole page. Offset is not impt
→ Replace page within that process

Page Replacement Algorithms: Evaluation

Memory access time:
$$T_{access} = (1-p) * T_{mem} + p * T_{page\_fault}$$
- p = probability of page fault
- T_{mem} = access time for memory resident page
- T_{page_fault} = access time if page fault occurs
- Since $T_{page\_fault} >> T_{mem}$
  - Need to reduce p to keep T_{access} reasonable
- See for yourself, try to find p if:
  - T_{mem} = 100ns, T_{page_fault} = 10ms, T_{access} = 120ns

Good algorithm should **reduce the total number of page faults**

→ Reduce probability of page fault to reduce access time.

Optimal Page Replacement (OPT)
- Theoretical, not possible to achieve in real life as we need future knowledge of memory references
- We can **guarantee optimum performance** (least number of page faults) if we **replace the page that will not be used again for the longest period of time**
- OPT is used as a base of comparison for other algorithms. The closer to OPT == better algorithm

Example: OPT (6 Page Faults)

Time	Memory Reference	Frame A	Frame B	Frame C	Next Use Time A	Next Use Time B	Next Use Time C	Fault?
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				

<u>FIFO Page Replacement Algorithm</u>
- Memory pages evicted based on their **loading time**, evict oldest memory page
Implementation: OS maintain a queue of resident page numbers, remove the first page in queue if replacement needed, update the queue during page fault trap
→ Simple to implement: no hardware support needed

Example: FIFO (9 Page Faults)

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

→ **Note:** Queue is updated at load time, not access time. (It is pretty hard for the OS to take the old entry in the queue and place it at the end again)

FIFO: Problems

→ If given more RAM (more physical frames), number of page faults should decrease, but FIFO violates this

→ Belady's Anomaly – increase in frames → increase in page faults

Reason: FIFO does not exploit **temporal locality**

LRU (Least Recently Used) Page Replacement

→ Makes use of temporal locality: Replace the page that has not been accessed in the longest time

→ Expect a page to be reused in a short time window. Conversely, if a page has not been accessed for some time, it most likely wont be accessed again soon

→ Attempts to approximate the OPT algorithm, gives good results generally, does not suffer from Belady's Anomaly

Example: LRU (7 Page Faults)

Time	Memory Reference	Frame A	Frame B	Frame C	Last Use Time			Fault?
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	

→ Not sure how the last use time is updated in terms of implementation, perhaps have an array mapping the page number to last updated timing, take O(N) time to evict since need to find least recently used

LRU: Implementation Details

→ Not easy: need to keep track of "last access time" somehow

Approach A: Use a counter

- A logical 'time' counter which is incremented for every memory reference
- Page table entry has a 'time of use' field, store time counter value whenever reference occurs
- Replace page with smallest 'time of use'. To do that need O(N) time to search through all the pages

→ 'Time of use' forever increasing, might **overflow**

Approach B: Use a "stack" (its more like a double ended queue)

- Maintain a stack of page numbers
- If a page X is referenced, remove from the stack (if entry exists), push on top of the stack
- Replace the page at the bottom of the stack (No need to search through all entries)

Problems: Not a real stack, hard to implement in hardware

Second-Chance Page Replacement (CLOCK)

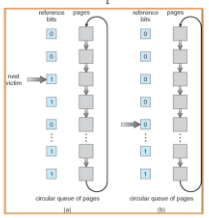
- Modified FIFO to give second chances to pages that are accessed
- Each PTE maintains a 'referenced' bit: 1=accessed, 0=not accessed

- Algorithm:
  1. The oldest FIFO page is selected
  2. If reference bit == 0 → Page is replaced
  3. If reference bit == 1 → Page is given a 2nd chance
     - Reference bit cleared to 0
     - Load time reset → page taken as newly loaded
     - Next FIFO page is selected, go to Step 2
- Degenerate into FIFO algorithm
  - When all pages have reference bit == 1 (or all == 0)

Second-Chance: Implementation Details

- Circular Queue

Second-Chance: Implementation Details



- Use **circular queue** to maintain the pages:
  - With a pointer pointing to the oldest page (the **victim page**)
- To find a page to be replaced:
  - Advance to a page with '0' reference bit
  - Clear the reference bit as pointer passes through

→ Once a page with '0' ref bit is found, replace on the spot?

Example: CLOCK(6 Page Faults)

Time	Memory Reference	Frame (with Ref Bit) A	B	C	Fault?
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)	

▶ Victim Page

Frame Allocation

- What is the best way to distribute the limited N frames to M processes each with P pages?

- Simple Approaches:
  - **Equal Allocation:**
    - Each process gets `N / M` frames
  - **Proportional Allocation:**
    - Processes are different in size (memory usage)
    - Let $size_p$ = size of process p, $size_{total}$ = total size of all processes
    - Each process gets $size_p/size_{total}*N$ frames

→ Local Replacement – Victim pages selected among pages of process that caused page fault

→ Global Replacement – Process P can take a frame from Process Q by evicting Q's frame during replacement

Local Replacement vs Global Replacement

Local Replacement Pros/Cons

Pros: Frames allocated to process constant → stable performance between multiple runs

Cons: If frames allocated to process not enough → hinder performance of a process

Global Replacement Pros/Cons

Pros: Allow dynamic self-adjustment between processes – process that need more frames can get from other

Cons: Badly behaved process can affect others, frames allocated to a process is different from run to run (performance is not stable), cascading thrashing (see below)

Frame Allocation and Thrashing

→ Insufficient physical frames: Thrashing can occur, **heavy I/O** to bring non-residents into RAM

→ Hard to find number of frames

→ If global replacement is used: **A thrashing process 'steals' pages** from other processes, causing other processes to thrash (**Cascading Thrashing)**

→ If local replacement is used: Thrashing can be limited to one process, but that single process can hog the I/O (because it is trying to promote pages into residency) and degrade the performance of other processes

How to find the right number of frames?

→ Set of pages referenced by a process is relatively constant in a period of time (aka locality)

- As time passes, set of pages (**working set**) can change

→ This set of pages can change as time passes

Example: Executing a function usually has memory references on local variables, parameters, code etc in that function.

After function terminates, references will change to another set of pages

Working Set Model

- Using observation on locality: in a new locality, a process will cause page faults for the set of pages (basically means when we change locality, page faults start to happen)
- Once we load a new set of frames, no/few page faults until process transits to new locality

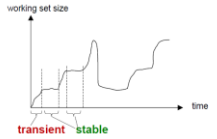
- Working Set Model:
  - Defines Working Set Window Δ
    - An interval of time
  - `W(t,Δ)` = active pages in the interval at time `t`
  - Allocate enough frames for pages in `W(t, Δ)` to reduce the number of page faults

→ Accuracy of working set model is directly affected by the choice of triangle.

Too small: miss pages in current locality

Too big: contain pages from different locality

Working Set Model: Illustration



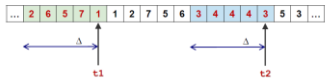
Transient region: working set changing in size

Stable region: working set about the same for a long time

- **Transient and stable regions**

Working Set Model: Illustration

- Example memory reference strings



- Assume
  - Δ = an interval of 5 memory references
- `W(t1,Δ)={1,2,5,6,7}` (5 frames needed)
- `W(t2,Δ)={3,4}` (2 frames needed)
- Try using different Δ values

- Triangle = 5

Pointers

- Page fault is checked by MMU (memory management unit, in the cpu i.e. hardware context), once page fault triggered, OS comes in to handle page fault and bring frame to phy mem (OS context). MMU comes in thereafter to check if page is in phy memory again

File System Introduction

- Resource management scheme, protection/sharing between processes and users

Criterias

1. Self-contained: Plug and play into another syst
2. Persistent: Beyond lifetime of OS and process
3. Efficient: Good management of free and used space, minimum overhead for bookkeeping

	Memory Management	File System Management
Underlying Storage	RAM	Disk
Access Speed	Constant	Variable disk I/O time
Unit of Addressing	Physical memory address	Disk sector
Usage	Address space for process **Implicit** when process runs	Non-volatile data **Explicit** access
Organization	**Paging/Segmentation:** determined by HW & OS	Many different FS: ext* (Linux), FAT* (Windows), HFS* (Mac OS)etc.

File

→ Can think of it as an abstract data type – interfaces a set of common operations with various possible implementations

→ Has data and metadata (aka file attributes)

File Metadata

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

Naming Rules:

- Common naming rule:
  - Length of file name
  - Case sensitivity
  - Allowed special symbols
  - File extension
    - Usual form `Name.Extension`
    - On **some** FS, extension is used to indicate **file type**

File Types:

- Each file type has:
  - An associated set of operations
  - Possibly a specific program for processing

- Common file types:
  - **Regular files:** contains user information
  - **Directories:** system files for FS structure
  - **Special files:** character/block oriented

Two Major Types of Regular Files

- **ASCII files:**
  - Example: text file, programming source codes, etc
  - Can be displayed or printed **as is**

- **Binary files:**
  - Example: executable, Java class file, pdf file, mp3/4, png/jpeg/bmp etc
  - Have a predefined internal structure that can be processed by specific program
    - JVM to execute Java class file
    - PDF reader for pdf file etc

Distinguishing File Types:

1. File type – description of info contained in file
2. Use file extension. Used by Windows, change of ext implies change in file type
3. Use embedded info in file: Used by Unix, **stored at beginning of file,** known as magic number

File Protection

Types of access: read, write, execute, append, delete, list (read metadata of file)

- We can restrict access of files based on user identity, using an **access control list**

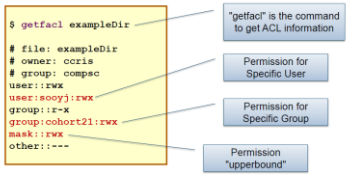
- **Access Control List**
  - A list of user identity and the allowed access types
  - **Pros:** Very customizable
  - **Cons:** Additional information associated with file

Permission Bits

- Classified the users into three classes:
  1. **Owner:** The user who created the file
  2. **Group:** A set of users who need similar access to a file
  3. **Universe:** All other users in the system
- Example (Unix)
  - Define permission of three access types (**R**ead/**W**rite/**E**xecute) for the 3 classes of users
  - Use "`ls -l`" to see the permission bits for a file

→ Bits 1-3: Owner, 4-6: group, 7-9: others
→ specifies what permissions are there for each type of user (read, write, execute)

- In Unix, Access Control List (ACL) can be:
  - Minimal ACL (the same as the permission bits)
  - Extended ACL (added **named users / group**)



Operations on File Metadata

- **Rename:**
  - Change filename
- **Read attribute:**
  - Get file creation time
- **Change attributes:**
  - File access permissions
  - Dates
  - Ownership
  - etc

File Data Structure

1. Array of bytes, each byte has offset (distance) from file start
2. Fixed length records: Array of records, can grow/shrink
   - Can jump to any record easily:
     - Offset of the Nth record = size of Record * (N-1)
3. Variable length records – Flexible, but harder to locate record

→ Records are ways to organise information of file and connect the info to the metadata of the file

File Data Access Methods

1. Sequential Access: Data read in order (like a linked list), starting from beginning. Cannot skip but can be rewound
2. Random Access: Data can be read in any order (like an array). Can be done using:
   1. `Read( Offset )`: Every read operation explicitly state the position to be accessed
   2. `Seek( Offset )`: A special operation is provided to move to a new location in file
   - E.g. Unix and Windows uses (2)

3. Direct Access

→ Allow random access to any **record** (not data) directly, i.e. if each record == one byte then it is also random access.

File Data Operations

File Data: Generic 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

File Operations as Syscalls

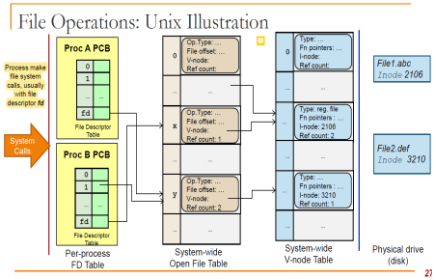
- Provides protection, concurrent and efficient access, maintains information
- Information kept for an opened file:
  - **File Pointer:** Current location in file
  - **Disk Location:** Actual file location on disk
  - **Open Count:** How many times has this file opened?
    - Useful to determine when to remove the entry in table

File Information in OS

- Several processes can open same file, several different files can be opened at any time
- Open – To add file into open file table and place file pointer at start of file. Close operation removes file from table of open files.

3 approaches:

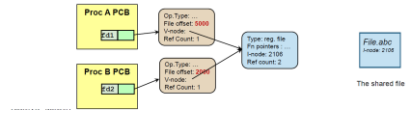
- Per-process open-file table:
  - To keep track of the open files for a process
  - Each entry points to the **system-wide open-file table** entries
- System-wide open-file table:
  - To keep track of all the open files in the system
  - Each entry points to **a V-node** entry
- System-wide V-node(virtual node) table
  - To link with the file on physical drive
  - Contains the information about the physical location of the file.



→ File offset belongs to system-wide file table
→ File opened twice from 2 processes can either have 1/2 separate system wide file table entries

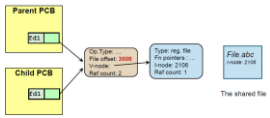
Process Sharing File in Unix: Case 1

- A file is opened twice from two processes:
  - 2 file descriptors
  - 2 entries in the system-wide open file table
  - I/O can occur at independent offsets
- When:
  - Two process open the same file
  - Same process open the file twice



Process Sharing File in Unix: Case 2

- Two file descriptors pointing to the same entry in the system-wide open file table
  - Only one offset → I/O changes the offset for the other process
- When:
  - fork() after file is opened
  - dup () within the same process



Directory

Possible Structures: Single-level, tree structure, directed acyclic graph, general graph

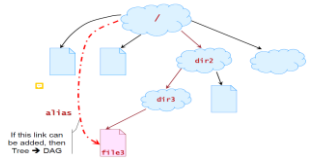
1. Single Level: One root directory storing all files

2. Tree-Structured: Directory storing files and directory (How we typically think of FS)

- Two ways to refers to a file:
  - **Absolute Pathname:**
    - Directory names followed from root of tree + final file
    - i.e. the Path from root directory to the file
  - **Relative Pathname:**
    - Directory names followed from the **current working directory (CWD)**
    - CWD can be set explicitly or implicitly changed by moving into a new directory under shell prompt

3. DAG: If file appears more than once in entire file system, there is only 1 copy of actual content, then files in various directories are simply pointers to one file

Directory Structure: DAG



→ Using **hard links and symbolic links** (**Note:** hard links are not allowed for directories!!)
→ Note: Symlink can cause cycles in DAG (symlink from A to B and B to A, but ls is well implemented to prevent infinite recursion)
→ We don't allow back links at all in DAG

Unix Hard Links

- **Consider:**
  - Directory A is the owner of file F
  - Directory B wants to share F
- **Hard Link:**
  - A and B has **separate pointers** point to the actual file F in disk
  - **Pros:**
    - Low overhead, only pointers are added in directory
  - **Cons:**
    - Deletion problems:
      - e.g. If B deletes F? If A deletes F?
      - Ref. count is needed
  - Unix Command: "`ln`"

→ inode will track how many references there are to it, if number of references reaches 0, it will drop the file from secondary storage and inode disappears

Unix Symbolic Links

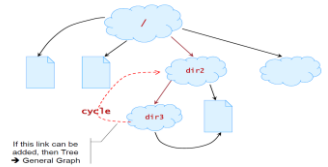
DAG: Unix Symbolic Link

- **Symbolic Link:**
  - The symbolic link is a ***special link file***, G
    - G contains the path name of F
  - When G is accessed:
    - Find out where is F, then access F
  - **Pros:**
    - Simple deletion:
      - If the symbolic link is deleted: G deleted, not F
      - If the linked file is deleted: F is gone, G remains (but not working)
  - **Cons:**
    - Larger overhead:
      - Special link file take up actual disk space
  - Unix Command: "`ln -s`"

→ **Note:** Symbolic Links use relative path (i.e. ln -s ../../topath directory/sl)

General Graph

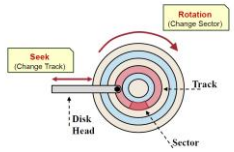
Directory Structure: General Graph



→ Not desirable (Hard to traverse, easy to infinite loop)
→ Hard to determine when to remove a file/directory
→ Symbolic links can create general graphs

I/O Scheduling

Magnetic Disk in One Glance



→ Seek and Rotation latency is high – OS should schedule disk I/O requests
Requirements: Reduce overall waiting time, reducing seeking time, balance need for high throughput while trying to fairly share I/O requests among processes

Disk Scheduling: Algorithms

- Consider the following disk I/O requests indicated by only the **track number (magnetic disks)**:
  - **13, 14, 2, 10, 17, 31, 21, 7**
- A few obvious candidates:
  - **FCFS**
  - **SSF** (**S**hortest **S**eek **F**irst)
    - "SJF" modified for the disk context
  - The **SCAN** family (aka **Elevator**):
    - Bi-Direction [Innermost ←→ Outermost] (SCAN)
    - 1-Direction [Outermost→ Innermost] (C-SCAN)

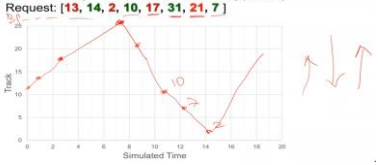
→ Tracks are floors in building, disk head is elevator servicing the floors, move 'on the way' to serve the requests.
→ SCAN – goes towards 0 then high numbers
→ CSCAN – goes towards high numbers, once reach limit, go back all the way to 0 and start again

Bi-Direction SCAN

→ From innermost to outermost, starting from the first request value (I THINK????)
→ Keep going 'up', since its on the way, then finally start going 'down'

SCAN: Disk Head Movement

- **Bi-Direction** [Innermost ←→ Outermost] (SCAN)
- Request: [13, 14, 2, 10, 17, 31, 21, 7]



→ Takes advantage of the way the disk moves and the head moves

Newer I/O Scheduling Algorithms

1. Deadline

– I/O Requests that come in are put into a sorted queue, depending on the type of storage device you use. Will minimize head movement of disk
- Also added to 2 other queues: Read FIFO and Write FIFO. These read and write requests are sorted chronologically in these queues
→ **If we want OS to be more responsive reading,** allocate more scheduling time to read queue, or set lower deadlines (vice versa)
→ If writes don't happen too often we can increase the deadline for writes (just delay them, they aren't that important anyway and they take longer than reading)

How does it work

- OS takes requests from sorted queue, at the same time keeping track of deadlines in FIFO queues. If operation's deadline is nearing then prioritize deadline.
- Other variations include having 2 sorted queues (1 for read/write each)

No-op

- Requests are just served as is, FIFO.

CFQ (Completely Fair queueing)

- Each process has sorted queue, with time slice allocated

BFQ (Budget Fair queueing)

- Fair sharing based on number of sectors requested. (I think if you request more, you will be put on lower priority)

Unix File Operations

- File related Unix System Calls
  - `open()`, `read()`, `write()`, `lseek()`, `close()`
- General Information:
  - Opened file has an identifier
    - **File Descriptor:** Integer
    - Used for other operations
  - File is access on a byte-by-byte basis
    - No interpretation of data

Open()

Opening Files: `open( )`

- Function Call:
  `int open( char *path, int flags )`
- Return:
  - `-1`: Failed to open file
  - `>=0`: **file descriptor**, a unique index for opened file
- Parameters:
  - `path`: File path
  - `flags`: Many options can be set using bit-wise-OR
    - Read, Write or Read+Write mode
    - Truncation, Append mode
    - Create file if no exists
    - ... Many many more ☺

```
//Open an existing file for read only
fd = open( "data.txt", O_RDONLY );

//Create the file if not found, open for read + write
fd = open("data.txt", O_RDWR | O_CREAT );
```

By convention:

- Default file descriptors:
  - STDIN (0), STDOUT (1), STDERR (2)

Read()

Read Operation: `read()`

- Function Call:
  `int read(int fd, void *buf, int n)`
- Purpose:
  - reads up to n bytes from current offset into buffer buf
- Return:
  - number of bytes read, can be `0...n`
  - `<n` : end of file is reached
- Parameters:
  - `fd`: file descriptor (must be opened for read)
  - `buf`: An array large enough to store n bytes
- `read()` is ***sequential read***:
  - starts at current offset and increments offset by bytes read

Write()

- Function Call:
  `int write(int fd, void *buf, int n)`
- Purpose:
  - `writes` up to n bytes from current offset from buffer buf
- Return:
  - **-1**: Error
  - **>= 0**: Number of bytes written
- Parameters:
  - `fd`: file descriptor (must be opened for write)
  - `buf`: An array of at least n bytes with values to be written
- `Possible errors:`
  - exceeds file size limit, quota, disk space, etc.
- write() is ***sequential write:***
  - starts at current offset and increments offset by bytes written
  - can increase file size beyond EOF → append new data

Lseek()

- Function Call:
  `off_t lseek(int fd, off_t offset, int whence)`
- Purpose:
  - Move current position in file by **offset**
- Return:
  - **-1**: Error
  - **>= 0**: Current offset in file
- Parameters:
  - `fd`: file descriptor (must be opened)
  - **offset**: positive = move forward, negative = move backward
  - **whence**: Point of reference for interpreting the **offset**
    - `SEEK_SET`: absolute **offset** (count from the file start)
    - `SEEK_CUR`: relative **offset** from current position (+/-)
    - `SEEK_END`: relative **offset** from end of file (+/-)
- Can seek anywhere in file, even beyond end of existing data

Close()

- Function Call:
  `int close( int fd )`
- Return:
  - **-1**: Error
  - **0**: Successful
- Parameters:
  - `fd`: file descriptor (must be opened)
- With close():
  - `fd` no longer used anymore
  - Kernel can remove associated data structures
  - The identifier `fd` can be reused later
- By default:
  - Process termination automatically closes all open files

File System Implementations

- General Disk structure: 1D array of logical blocks, logical block mapped to disk sector, layout of disk is hardware dependent

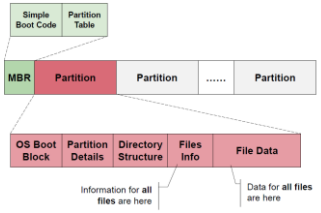
Disk Organization

- MBR (Master Boot Record) at sector 0 with partition table, followed by 1 or more partitions.
- Each partition is an independent file system

File System contains:

1. OS Boot up Information
2. Partition Details: Total number of blocks, number and location of free disk blocks
3. Directory Structure
4. Files information
5. Actual File Data

Generic Disk Organization: Illustration



→ May differ for different implementations.

Implementing File

- Collection of logical blocks
- When fileSize != multiple of logical blocks, we get internal fragmentation, last block has wasted space.

→ Should keep track of logical blocks, allow efficient access and utilize disk space effectively

- Focus on how to allocate file data on disk

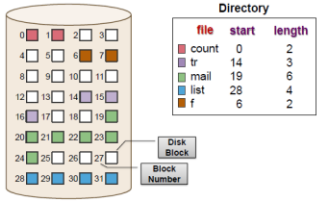
File Allocation 1: Contiguous

→ Allocate consecutive disk blocks to a file

Pros: Easy to track, just need to track starting block number + length. Fast access (only need to seek for first block)

Cons: External Fragmentation. With multiple file creation/deletion, disk can have small holes. File size need to be specified in advance.

Contiguous Block Allocation



→ Why do we need len attribute? We're storing in an 'array', so need length to know which is last block

File Block Allocation 2: Linked List

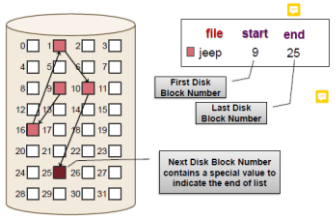
General Idea: Keep Linked list of disk blocks.

Each disk block stores: Next disk block number (pointer), actual file data.

File information stores: first and last disk block number

Why do we need last disk block? We can make the last block point to a null block to indicate eof, but a disk block occupies a large space, so it is very wasteful to use it to represent eof. Better to store last block location inside file info table.

Linked List Allocation



→ **Issue:** Very hard to traverse to a file's offset, need to traverse the file from the start

File Block Allocation 2: Linked List V2.0

General Idea: Move all the block pointers into a single table, known as a FAT table.

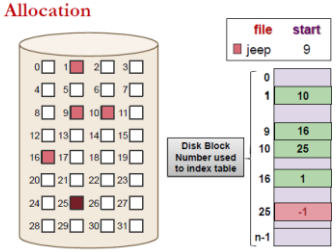
→ FAT is always in memory.

→ Simple and efficient, used by MS-DOS

Pros: Faster random access, linked list traversal takes place in memory

Cons: Keeps track of all disk blocks in a partition, the FAT table can be huge when the disk is large, consuming valuable memory space

FAT Allocation



File Block Allocation 3: Indexed Allocation

General Idea: Each file has an index block. Basically we have an array of disk block addresses **that is stored in some disk block**

→ Array of disk block addresses. IndexBlock[N] == Nth Block Address

Pros: Lesser memory overhead. Only index block of opened file needs to be in memory. Fast Direct Access

Cons: Limited max file size. Max number of blocks == number of index block entries. Index block overhead

Free Space Management

- Need to know which disk block is free to perform file allocation, i.e. maintain a free space list

Free space management: Maintain free space information – **Allocate** (Remove free disk block from free space list, when files are created or enlarged (appended)), **Free** (Add free disk block to free space list, when file is deleted or truncated)

Free Space Management: Bit Map

- Each disk block represented by 1 bit: 1 == free, 0 == occupied
- Example:

`0 1 0 1 1 1 0 0 1 0 1 1 .....`

  - Occupied Blocks = 0, 2, 6, 7, 9, ...
  - Free Blocks = 1, 3, 4, 5, 8, 10, 11, ...

Pros: Provide a good set of manipulations. E.g. Can find the first free block, n-consecutive free blocks easily by performing bitwise operations.

Cons: Must be kept in memory for efficiency reasons

Free Space Management: Linked List

Use linked list of disk blocks. 2 ways to implement:

1. Have each disk block contain a list of free disk blocks. Persistent inside disk block. We will also need a pointer to this specific disk block
2. For each free disk block, have a pointer to the next free space disk block. Incurs overhead within free disk blocks.

Pros: Easy to locate free block.

Only the first pointer to a free block is needed in memory. Other blocks can be cached if wanted for efficiency

Cons: High Overhead. Can be mitigated by storing the free block list in free blocks

Implementing Directory

1. Keep track of files in a directory, have some file metadata, map file name to file info

Directory Structure: We are given full path name, need to recursively search directories along the path to arrive at the file info

- Example:
  - Full path name: `/dir2/dir3/data.txt`
  1. Find "`dir2`" in directory "/"
     - Stop if not found (or incorrect type)
  2. Find "`dir3`" in directory "`dir2`"
     - Stop if not found (or incorrect type)
  3. Find "`data.txt`" in directory "`dir3`"
     - Stop if not found (or incorrect type)
- Sub-directory is usually stored as file entry with special type in a directory

(Directory is a type of file itself)

Directory Implementation: Linear List

- Directory consists of a list, each entry represents a file. **Store** file name and possibly other file metadata, **store** file information or pointer to file information

Locate a file using list: Requires linear search, O(N). Inefficient for large directories and/or deep tree traversals.

Solution: Use cache to remember latest few searches, user moves up/down a path

Directory Implementation: Hash Table

- Each directory contains a hash table of size N
- Locate a file by filename: File name is hashed into index K between 0 and N – 1
- HashTable[K] inspected to match file name.

Chained collision resolution is used, i.e. file names with same hash value chained together in LL.

Pros: Fast lookup

Cons: Hash table has limited size, depends on good hash function

Directory Implementation: File Information

File Information consists of:

1. Filename and other metadata
2. Disk block information

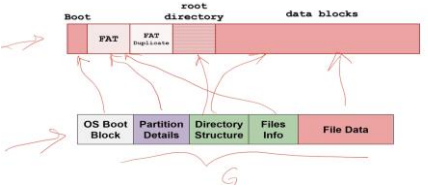
2 common ways to implement:

1. Store everything in directory entry. Simple scheme is to have fixed size entry, i.e. all files have same amount of space for information.
2. Store only file name and points to some data structure for other info (**in ext2,** filename stored in directory, points to inode, inode points to real data)

FAT File System

FAT12 → FAT 16 → FAT32 (32 means each fat entry is 32 BITS, not bytes)

File System Layout

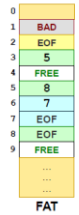


File Data
- Allocated to a number of data blocks/data block clusters
- Allocation info is kept as a linked list (All data block pointers kept separately in the FAT table)

FAT (File Allocation Table):
- One entry per data block/cluster
- Store disk block info (Free? Next block (if occupied)? Damaged?
- **OS will cache FAT in RAM to facilitate linked list traversal**

FAT Allocation Table: Illustration

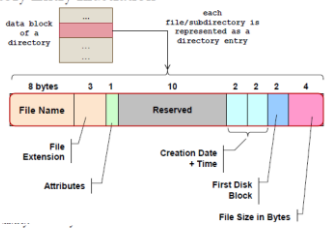
- FAT entry contains either:
  - FREE code (block is unused)
  - Block number of next block
  - EOF code (i.e. NULL pointer)
  - BAD block (block is unusable, i.e. disk error)
- Example:
  - Block 3 → 5 → 8 → EOF
  - Block 4 and 9 are free
  - Block 1 is unusable



Directory Structure and File Information
Directory is represented as:
- Special type of file
- Root directory stored in a special location, other directories stored in data blocks.
- Each file/subdir within the folder represented as directory entry

Directory Entry:
- Fixed-size 32-bytes(????) per entry
  - Contains:
    - Name + Extension
    - Attributes (Read-Only, Directory/File flag, Hidden etc)
    - Creation Date + Time
    - First disk block + File Size

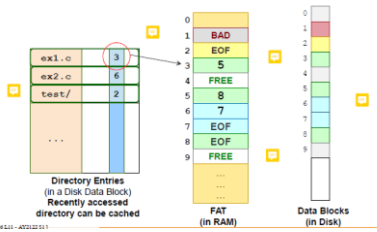
Directory Entry Illustration



- File Name + Extension
  - Limited to 8+3 characters
  - The first byte of file name may have special meaning:
    - Deleted, End of directory entries, Parent directory, etc.
- File Creation Time and Date:
  - Year is limited to 1980 to 2107
  - Accuracy of second is ±2 seconds
- First Disk Block Index:
  - Different variants uses different number of bits:
    - 12, 16 and 32 bits for FAT12, FAT16 and FAT32 respectively

→ First Disk block idx: 1st data block of file

FAT FS: Putting the parts together…



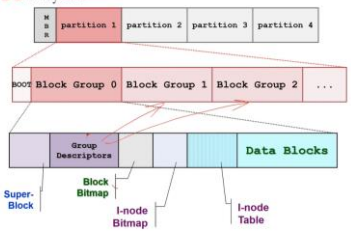
→ **We traverse within the data block to find an entry. Each directory entry is 32 bytes. In FAT, to find free space,** we need to traverse the FAT file to find free disk blocks. (Slow because traversal, but remember FAT is always in RAM so not that slow). Each FAT entry is 4 bytes.
Algo to read file: check first disk block index, access index on disk, access index on FAT, acces 5 on disk, access 5 on FAT...

Notes about FAT16/32
- we can only store 2^16 blocks worth of data in the FAT16 partition.
- Use first disk block number stored in directory entry to find starting point of linked disk blocks
- Use FAT to find out the subsequent disk block number, terminated by EOF
- Use disk block number to perform disk access on data blocks
- Directories are stored is disk blocks. Contains directory entries.

Ext2 (Extended 2 File System)
- Split disk space into Blocks
→ Similar to disk cluster in FAT FS
- Blocks are grouped into Block Groups

Each file/directory is described by:
- I-Node, which contains file metadata (access right, creation time) and data block addresses

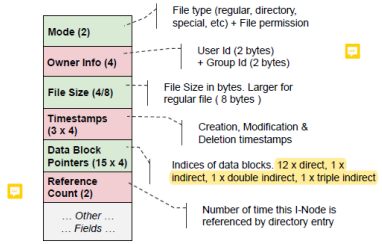
Ext2 FS: Layout



Partition Information
Superblock – Describes the whole file system, includes: total inode number, inodes per group, total disk blocks, disk blocks per group

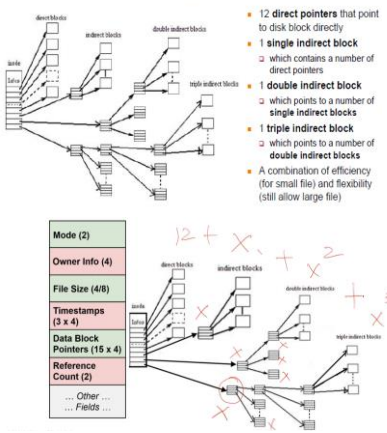
Redundancy – Information is duplicated in each block group for redundancy
Group Descriptors – Describes each of the block group (Number of free disk blocks, free inodes, location of bitmaps). **Duplicated** in each block group as well.
Block Bitmap – Keeps track of the usage status of blocks of this block group (1 = occupied, 0 = free)
I-Node Bitmap – Keep track of the usage status of I-Nodes of this block group (1 = occupied, 0 = free)
I-Node table – An array of I-Nodes, each i-node can be accessed by a unique index. Contains only i-nodes of this block group. I-Node table is on disk, when opened, add entry to open file table (in memory), and entry to in-memory inode table (cached for easy reference). Inode structure

Ext2: I-Node Structure (128 Bytes)



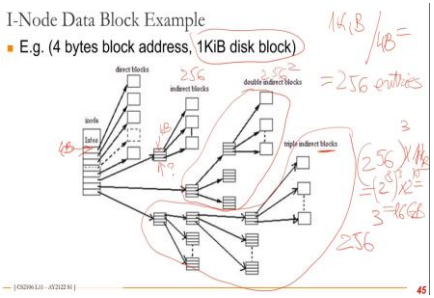
Multilevel Data Blocks
- Ext2 stores their file data blocks in a multi-layered fashion, similar to multi-level paging
→ Allows for larger file size
→ Combination of direct indexing and multi-level index scheme
→ 12 direct pointers, 1 single indirect, 1 double indirect which points to single indirects, 1 triple indirect which points to double indirects

I-Node Structure: Data Blocks



→ if the number of entries in each block is x, then single indirect can have x entries, double indirect has x^2, triple indirect has x^3 entries

I-Node Data Block Example

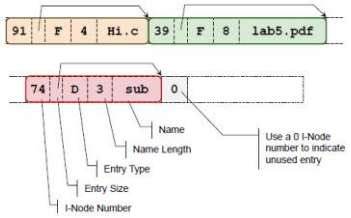


→ Example: 4 bytes addresses, 1KB disk block. => 256 entries per block. From triple indirect table, 256^3 * 1KB = 16GB.
Note: Each entry stores an address, not the data itself! The data is pointed to by the address specified in the block.

Directory Structure
- Directory also stored in data blocks, each entry is of variable length, stored as a LL.
- Each directory entry contains:
  - I-Node number for that file/subdirectory
  - Size of this directory entry
    - For locating the next directory entry
  - Length of the file/subdirectory name
  - Type: File or Subdirectory
    - Other special file type is also possible
  - File/Subdirectory name (up to 255 characters)

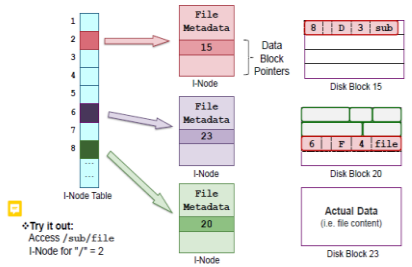
→ Each directory entry has variable length

Directory Entry (Illustration)



Ext2 Putting it together

Ext2: Putting The Parts Together

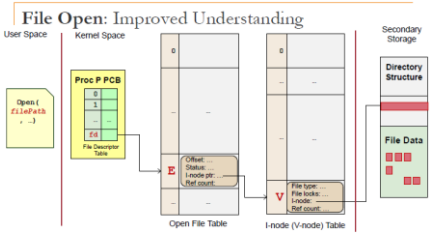


→ To access directory: Start with root at inode 2. Use inode table (inodeTable[2]) to access inode, then access data block pointer to go to directory entries, check inode-table 8, go to disk block 20, go to inode 6 via inode table, access 23 to get actual data

Walkthrough on file operation: Open()
- Process P open file /.../.../.../F:
  - Use full pathname to locate file F
    - If not found, open operation terminates with error
    - When F is located, its file information is loaded into a new entry E in system-wide table
    - Creates an entry in F's table to point to E (file descriptor)
    - Return the file descriptor of this entry
- The returned file descriptor is used for further read/write operation

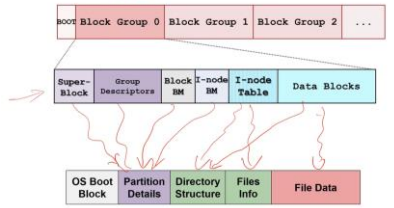
→ Basically locate file, then add pointer to file's inode in open file table, indexed by fd

File Open: Improved Understanding

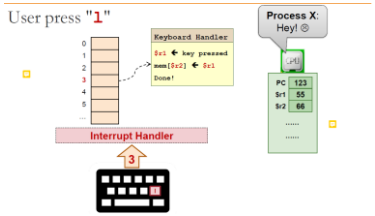


Ext2 Other common tasks (delete file)
- Deleting a file:
  - Remove its directory entry from the parent directory:
    - Point the previous entry to the next entry / end
    - To remove first entry: Blank record is needed
  - Update I-node bitmap:
    - Mark the corresponding I-node as free
  - Update Block bitmap:
    - Mark the corresponding block(s) as free
- Question:
  - Is it possible to "undelete" a file under ext2?
  - What if the system crashes between the steps?

Stop & Check: Ext2



OS Summary
Interrupts – different interrupts have some value mapped to it, whereby the OS has a list of function pointers. OS uses the value mapped to trigger the interrupt handler
→ In order to execute handler, we need to change states inside our registers, so we save the state of the process inside the PCB, and then execute the interrupt handler. Basically do a 'mini' context switch, since its only swapping out of registers



Interrupt steps

- Give the sequence of steps for handling an interrupt.
  - Interrupt occurs
  - Save registers/CPU state
  - Perform the handler routine
  - Restore registers/CPU state
  - Return from interrupt

songy into pcb

Memory Context

→ Inside PCB, we are storing **POINTER** to the page table and TLB entries. NOT the entries themselves

Copy on Write when fork()

- We do not copy frames when we fork. We duplicate the page table. When a write occurs (in either parent or child), an interrupt happens which triggers the OS to write contents to a fresh physical frame and update contents. We also update page table entry info.

Inode

- You can access inodes from any group, not just your own group!!!
- Each inode has an identifier, can identify which group an inode is from based on its id.

→ Any inode in the filesystem can be accessed from any other group.

Incorrect implementation of General Semaphore using Binary Semaphore

(General semaphores) We mentioned that a general semaphore (S > 1) can be implemented by using one or more **binary semaphores** (S == 0 or 1). Consider the following attempt:

```
int count = <initially: any non-negative integer>;
Semaphore mutex = 1; // binary semaphore
Semaphore queue = 0; // binary semaphore, for blocking tasks
```

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue)
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

ANS:

a. The issue is task interleaving between "`signal(mutex)`" and "`wait(queue)`" in `GeneralWait()` function. Consider the scenario where count is 0, two tasks A and B execute `GeneralWait()`, as task A clears the "`signal(mutex)`", task B gets to executes until the same line. At this point, count is -2. Suppose two other tasks C and D now executes `GeneralSignal()` in turns, both of them will perform `signal(queue)` due to the count -2. Since `queue` is a binary semaphore, the 2nd `signal()` will have undefined behavior (remember that we cannot have S = 2 for binary semaphore).

Correct Implementation:

```
int count = <any non-negative integer>;
Semaphore mutex = 1; // binary semaphore
Semaphore queue = 0; // binary semaphore, for blocking tasks
```

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } // else removed
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    } else { // else added
        signal(mutex);
    }
}
```

Using Pipes

```
527         int pipeFd[2];
528         char buffer[100];
529         pipe(pipeFd);
530         int childResult = fork();
531         if (childResult != 0) {
532             int exitStatus;
533             close(pipeFd[WRITE_END]);
534             int len = read(pipeFd[READ_END], buffer, sizeof(buffer));
535             close(pipeFd[READ_END]);
536             wait(&exitStatus);
537
538             FILE *outputPtr = fopen(fileName, "w");
539             for (int i = 0; i < len; i++) {
540                 fputc(buffer[i], outputPtr);
541             }
542             fclose(outputPtr);
543
544             if (isStdErr == 1) {
545                 printf("hello");
546                 *isFailedCommandPtr = 1;
547                 exit(2);
548             }
549
550             if (WEXITSTATUS(exitStatus) != 0) {
551                 *isInvalidCommandPtr = 1;
552                 exit(exitStatus);
553             }
554             exit(0);
555         }
```

MIPS

lw <into this register> <offset>(value in this reg)
sw <value in this reg> <offset>(to this register)
add <save here> <register> <register>
addi <save here> <register> <integer>

Pipe based lock

```
/* Define a pipe-based lock */
struct pipelock {
    int fd[2];
};

/* Initialize lock */
void lock_init(struct pipelock *lock) {

    pipe(lock->fd);
    write(lock->fd[1], "a", 1);

    // The first write is meant to initialize the lock such that exactly one thread can acquire the lock.

}

/* Function used to acquire lock */
void lock_acquire(struct pipelock *lock) {

    char c;
    read(lock->fd[0], &c, 1);

    // read will block if there is no byte in the pipe.

    // Closing the reading or writing end of the pipe in a thread will cause closing that end for all threads of the process (shared variable). Also, it will prevent all other threads to acquire or release the lock.
}

/* Release lock */
void lock_release(struct pipelock * lock) {

    write(lock->fd[1], "a", 1);

    // Need to write/read exactly one byte to simulate increment/decrement by 1 of a semaphore. It might work with multiple bytes, but you need to take care how many bytes are read/written.

}
```

Dining Philosopher – One Right handed

5. (Dining Philosophers) Our philosophers in the lecture are all left-handed (they pick up the left chopstick first). If we force **one of them** to be a right-hander, i.e. pick up the right chopstick before the left, then it is claimed that the philosophers can eat without explicit synchronization. Do you think this is a **deadlock-free solution** to the dining philosopher problem? You can support your claim informally (i.e. no need for a formal proof), but it must be convincing ☺.

ANS:

The claim is TRUE. Informal argument below.

For ease of discussion, let's refer to the right-hander as **R**.

If **R** grabbed the right chopstick then managed to grab the left chopstick THEN
R can eat → not a deadlock.

If **R** grabbed the right chopstick but the left chopstick is taken THEN
The left neighbor of R has already gotten both chopstick s → eating → eventually release chopsticks.

If **R** cannot grab the right chopstick THEN
The right neighbor of R has taken its left chopstick. Worst case scenario: all remaining left-handers hold on to their left chopsticks. However, the left neighbor of R will be able to take its right chopstick because R is still trying to get its right fork.

How does OS provide abstraction/protection for process(CPU time), memory and file

Process Management

Abstraction: illusion that process executes on CPU all the time.

Protection: Execution context of each process is isolated from each other.

Memory Management

Abstraction: illusion that process owns the entire memory space.

Protection: Memory space of each process are mapped to different physical address, isolating them from each other.

File Management

Abstraction: Files is a single contiguous logical entity.

Protection: Files can only be opened through system call, OS can prevent files from being opened for incompatible operations.

Implementing a Barrier with Semaphores

Questions for your own exploration

6. (Semaphore) In cooperating concurrent tasks, sometimes we need to ensure that **all N tasks** have reached a certain point in the code before proceeding. This specific synchronization mechanism is commonly known as a **barrier**. Example usage:

```
//some code

Barrier(N);  // The first N-1 tasks reaching this point
             // will be blocked.
             // The arrival of the Nth task will release
             // all N tasks.

// Code here only gets executed after all N tasks
// reach the barrier above.
```

Use **semaphores** to implement a **one-time use `Barrier()` function without using any loops.** Remember to indicate your variables declarations clearly.

```
int arrived = 0; // shared variable
Semaphore mutex = 1; // binary semaphore to provide mutual exclusion
Semaphore waitQ = 0; // for N - 1 processes to block on


Barrier(N) {
    wait(mutex);
    arrived++;
    signal(mutex);
    if (arrived == N)
        signal(waitQ);

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

This is not a **reusable barrier**, as `waitQ` may have undefined value afterwards, e.g. if tas interleave just before `if (arrived == N)`. multiple tasks can execute the fi `signal(waitQ)` resulting in `waitQ > 0` at the end. Note that this does not affect correctnes

OS mechanisms for dynamic allocation for memory

5. (Dynamic Allocation, adapted from [SGG]) It is possible for a program to dynamically allocate (i.e., enlarge the memory usage) during runtime. For example, the system call `malloc()` in C or **new** in Java/C++ can enlarge the **heap region** of process memory. Discuss the OS mechanisms needed to support dynamic allocation in the following schemes:
   a. Contiguous memory allocation (both fixed and dynamic size partitioning)
   b. Pure Paging
   c. Pure Segmentation

ANS:

a. It is simpler to have the heap region to be allocated at the end of the logical memory space. Then, we can enlarge the heap region by enlarging the partition allocated to the process.

Under fixed partitioning:
- No extra work is needed, as heap region can simply use up the free space (the internal fragmentation) between the partition size and the actual memory size

Under dynamic partitioning:
- If the adjacent partition is free:
  - Simple: Simply modify the partition information, i.e. change the length of current partition and shorten the free partition.

- If adjacent partitions are occupied:
  - More troublesome: The current partition cannot be enlarged. Relocation is required. OS need to look for a large enough free partition to fit the enlarged partition. Once located, the current partition is moved over.

b. Due to internal fragmentation of the paging scheme, it is possible that the allocation can use the remaining free space in the page. Suppose the allocation overshoot the page boundary, then OS needs to look for a free physical frame *f*. Afterward, update the page table by changing the first invalid page table entry from invalid to valid and fill in frame number *f*.

c. Idea is similar to the dynamic partitioning. If there is free memory at the end of the heap segment, then OS can simply update the limit of the segment and reduce the size of the affected free partition. If there is no free memory, then relocation is required. After relocation, both base and limit of the heap segment needs to be updated.

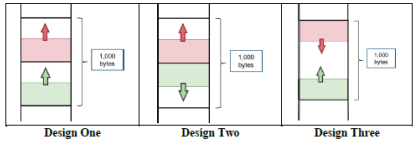
Buddy System Max/min fragmentation

6. Calculate the minimum and maximum percentage of memory capacity lost to internal fragmentation in a system that uses the Buddy allocator. Can the Buddy allocator suffer from external fragmentation?

ANS:
For an allocation request of N bytes: min fragmentation = 0% (exact fit, N=2^k), max fragmentation = ~50% (N=2^(k-1)+1) Note that internal fragmentation >50% is not possible.
External fragmentation can still happen!

Stack/Heap Design

7. (Growing/Shrinking of 2 Regions) This question looks at the problem of maximizing the *logical memory space* for two growing/shrinking regions (e.g., Stack and Heap regions). Suppose we have only a piece of 1,000 bytes of memory space. Which of the following placements of the stack and heap regions is the best choice? The arrows represent the growing direction of the regions. Briefly justify your choice:



ANS:
Designs #1 and #2 place a pre-defined upper limit on the two regions. These upper limits may not make sense for certain execution behaviour. You can easily find execution scenario that utilize heavily one of the regions over the other (e.g., heavy recursive function calls → stack region may be exhausted, heavy use of malloc → heap region may be exhausted instead). In such cases, one of the regions may reached the upper limit even though there may still be free memory space.

Design three allows free growing of the two regions. The regions reach their maximum size only when the growing boundary of the two regions meet, i.e. the memory space is fully utilized. This is the most commonly seen memory layout (heap and stack grows toward each other).

Algorithm for accessing virtual addresses VA / Page Fault Handler

Solution

a. **Algorithm for accessing virtual address VA:**
   1. [HW] VA is decomposed into <Page#, Offset>
   2. [HW] Search TLB for <Page#>:
      a. If TLB miss: Check Page table
   3. [HW] Is <Page#> memory resident?
      a. Non-memory resident: Trap to OS { Page Fault }
   4. [HW] Use <Frame#><Offset> to access physical memory.

Algorithm for Page Fault Handler, given <Page #>:
1. [OS] Global/Local replacement, Replacement algorithm applicable here
2. [OS] Write out the page to be replaced if needed.
3. [OS] Locate the page in secondary swap pages.
4. [OS] Load the page into a physical frame.
5. [OS] Update page table entries.
6. [OS] Flush TLB if needed.
7. [OS] Return from Trap

Calculating Overhead for Page Fault (Direct Paging)

TLB access: 1ns, Mem access: 30ns, Disk access: 5ms

Best case: TLB-hit: TLB access + 1 mem access = 31ns

Worst Case: Non-resident + eviction: TLB access + 1 mem access (to check pte) + service page fault (write out new page, write in old page) = 1ns + 30ns + 5ms + 5ms = 10,000,031ns

Worst Case for 2-level paging: TLB access + 1 mem access to find page table, but its in disk (page fault) + access page table to find frame mapping, but its also in disk

Total: 1 TLB access + 3 mem access + 2 page faults = 1ns + 90ns + 40ms

Question about demand paging (basically the inspiration of lab4)

Part c about the advantages and disadvantages of memory-mapped files compared to regular read/write might be important

4. (Applications of virtual memory)
An application of virtual memory in some OSes (including Linux) is overcommit. When overcommit is enabled, the OS will not allocate pages immediately upon request, but instead only when they are actually used.

In Linux, a new memory allocation has its contents initialised to zero. Overcommit allows Linux to defer allocating pages for a memory allocation until the program writes to the memory, (if it ever does).

a. How can this be implemented, while allowing reads to be successful?

Another application of virtual memory is demand paging, which ties in closely with the idea of memory-mapped files (i.e. `mmap`). Memory-mapped files are a natural extension to the idea of a swap file; in a way, we are simply allowing programs to specify the "backing store" for a particular region of memory.

b. How can demand-paged (i.e. the file is not read until the program actually reads from the mapped region) memory-mapped files be implemented?
c. What are the advantages and disadvantages of using memory-mapped files compared to regular read/write system calls?

Solution

a. Allocate a single read-only "zero page" that is filled with zeroes. When a program requests for memory, map the new pages to the zero page as a read-only mapping. If the program writes to memory, a segmentation fault/general protection fault will be triggered; at this point, we have to actually allocate a page to the program.
b. When a program maps a file, simply record the memory region as mapped to the given file, and insert the relevant page table entries as non-resident pages. (Some other tracking structures will likely be needed.) When the program tries to read from the region, a page fault will occur, and the OS can then perform the file read for the page at that point.
c. Advantages:
   i. (For the programmer) The programmer can simply read/write from memory and not have to deal with read/write syscalls.
   ii. (For the OS) The OS can easily reclaim those pages in physical memory by simply reusing the pages (if it knows that the pages have not been modified), or by writing the changes back to the file (which can be done even if there is no swap file configured on the system), since it can simply read the pages from the file again if a program reads from the mapped region of memory.
   iii. (For the OS/system as a whole) With read/write syscalls, the data is read into memory that is private to the program. If multiple processes read from the same file, the data is duplicated in memory. With memory-mapped files, the OS can simply share the same physical pages across multiple processes, saving physical memory.
d. Disadvantages:
   i. It is generally not possible to return an error from a memory read. If there is an error when reading the mapped file to fulfil a memory read, the program gets a SIGSEGV or SIGBUS, which is much harder to recover from than a failed syscall.
   ii. It is also not possible to fulfil memory reads asynchronously. Thus if the file read takes a while, the program will be blocked until the read completes (or fails). (This can be slightly mitigated with the madvise syscall.)

Maximum possible addressable physical memory

→ Based on the assumption that 1 bit can represent one frame in a PTE. (part b)
→ Consider using 1-level or 2-level paging based on the overhead of the page tables.

5. (Page table structure)
A computer system has a 36-bit virtual address space with a page size of 8 KB, and 4 bytes per page table entry.

a. How many pages are there in the virtual address space?
b. What is the maximum size of addressable physical memory in this system?
c. If the average process size is 8 GB, would you use a one-level or two-level page table? Why?

Solution

a. A 36-bit virtual address can address 2^{36} bytes in a byte addressable machine. Since the size of a page is 8 KB (2^{13}), the number of addressable pages is $2^{36} / 2^{13} = 2^{23}$.

b. Assuming that all 4 bytes in each PTE is used for the frame address, then we can address 2^{32} physical frames. Since each frame is 2^{13} bytes long, the maximum addressable physical memory size is $2^{32} * 2^{13} = 2^{45}$ bytes (32 TB).

c. 8 GB = 233 bytes. We need to analyse the memory and time requirements of the two paging schemes in order to make a decision. The average process size is considered in the calculations below.

1-level paging: Since we have 2^{23} pages in each virtual address space, and we use 4 bytes per page table entry, the size of the page table will be $2^{23} * 2^2 = 2^{25}$ (32 MB). This is 1 / 256 of the process's own memory space, so it is quite costly.

2-level paging: The single page table will be broken into multiple page tables, and each can fit into a single page. Since we have 8 KB (2^{13} B) pages and 4-byte PTEs, each small table can hold 2^{11} PTEs. There is a total of $2^{23} / 2^{11} = 2^{12}$ page tables → the page directory has 2^{12} entries. Hence, the virtual address would be divided up as 12 | 11 | 13.

Since the process' size is only 8 GB (2^{33}) → only $2^{33} / 2^{13} = 2^{20}$ pages. So we only need $2^{20} / 2^{11} = 2^9$ page tables (instead of the full range of 2^{12}).

The total overhead is then (size of page directory) + (total size of page tables) = $(2^{12} * 4) + (2^9 * 2^{11} * 4) = 4112$ KB.

As seen, 2-level paging requires much less space than level 1-paging scheme.

Calculating total overhead needed

→ Each process will have its own page table / page directory (because context switch) in direct and 2-level paging, but it is shared in inverted table.

- If direct paging, we calculate space with respect to entire virtual memory address space (for e.g. 32 bits -> 2^{32}/4KB = 2^{20} pages * size(pte) = 4 bytes = 2^{22}B = 4MB

If 2-level paging

1. always start with size of each page table = size of page.
2. Calculate number of entries per table and total entries in entire virtual memory address space.
3. Calculate number of page tables. Each page table takes one entry in page dir table. Calculate the number of page tables needed depending on the number of pages needed by process in qn

Open / close / truncating file

Purpose of open: Locate the file information using pathname + filename

d. What does it mean to open and close a file?
Operating systems keep a table of currently open files. The open operation enters the file into this table and places the file pointer at the beginning of the file. The close operation removes the file from the table of open files.

e. What does it mean to truncate a file?
Truncating a file means that all the information on the file is erased but the administrative entries remain in the file tables. Occasionally, the truncate operation removes the information from the file pointer to the end.

How do file permissions affect the actions allowed for a directory?

a. Perform "`ls -l DDDD`".
b. Change into the directory using "`cd DDDD`".
c. Perform "`ls -l`".
d. Perform "`cat file.txt`" to read the file content.
e. Perform "`touch file.txt`" to modify the file.
f. Perform "`touch newfile.txt`" to create a new file.

Can you deduce the meaning of the permission bits for directory after the above? Can you use the "directory entry" idea to explain the behavior?

ANS:

	ReadExeDir	WriteExeDir	ExeOnlyDir
a	ok	nope	nope
b	ok	ok	ok
c	ok	nope	nope
d	ok	ok	ok
e	ok	ok	ok
f	nope	ok	nope

→ ReadExeDir: chmod 500, WriteExeDir: chmod 300, ExeOnlyDir: chmod 100

→ ls -l DDDD is the same as calling ls -l within that directory.

→ We take a directory as a file with directory information

- **ls** → reading directory (file) contents, so need read permission
- **cd** → going into directory, need execute permissions.
- **cat** → if you can cd into file, you can cat file as long as read permissions permitted on the file
- **touch** → if used to edit file, just need file to have write permissions. If used to create file, need directory to have write permissions (since we are adding a new entry to the list for the dir and write isn't allowed)

a. Directory permission is independent from the file permission. So, you still can modify a file under a "read only" directory if the file allows write.

b. If you want to allow outsider to access a particular deeply nested file, e.g. A/B/C/D/file, you **only need** execute bit on A, B, C, D directory (i.e. read permission is not needed). This is a great way to hide the content of the directory and only allow access to specific file given the full pathname.

Idea of permissions for directory:

Then, the permission can be understood as:

Read = Can you read this list? (impact: ls, <tab> auto-completion)

Write = Can you change this list? (impact: create, rename, delete file/subdir). Note the interaction with the execute permission bit.

Execute = Can you use this directory as your working directory? (impact: cd).

Why do we have a 'open' syscall rather than passing a path to read and write syscalls?

3. (Adapted from [SGG]) Why do many operating systems have a system call to "open" a file, rather than just passing a path to the read or write system calls each time?

Answer:

- Either the read/write system calls return a handle to a file descriptor (like open normally does), or they do not and we pass a path to read/write each time.
- In the former case, it would complicate the interface of the system calls, because they would then need to also accept a handle instead of a path somehow.
- In the latter case, we would incur the cost of permission checking and path resolution each and every time the system call is made, which are non-trivial and expensive. Also, we lose the ability to have the OS track our offset within the file.
- In either case, such an interface would not generalise well to file descriptors that do not come from paths on the filesystem, like pipes and anonymous sockets.
- Some history: open, read, write and close were present in the original Unix OS from the start. Although pipes and sockets, etc., didn't come around until much later, the design generalised well to those. See

→ Aside of overhead, think of how pipes / stdin / stdout have fd values as well. 100 file ops

This leads to a strange phenomenon: it is generally true that the total time to perform 100 file operations for 1 item each is **much longer** than performing a single file operation for 100 items instead. e.g. writing one byte 100 times takes longer than writing 100 bytes in one go.

Benefit of buffered file operations

a. (Generalization) Give one or two examples of buffered file operations found in your favorite programming language(s). Other than the "chunky" read/write benefit, are there any other additional features provided by these high-level buffered file operations?

ANS:
a.
C: printf, scanf, fprintf, fscanf (printf and scanf are specifically to stdout and stdin, and are specialized versions of fprintf and fscanf. One can specify stdout/stdin as the file pointers. All of these are buffered. The buffer is flushed when (i) full or (ii) fflush is called or (iii) new line character is read/written)
Java: BufferedReader/BufferedWriter, Scanner. (BufferedReader/BufferedWriter have 8kB buffer while Scanner has 1kB)
C++: "<<", ">>" stream operators. Buffered.
Common additional features: error checking, packing/unpacking of datatypes (e.g. low level file operations are usually operating in bytes, it is useful to be able to directly operate on other datatype (int, float, etc) without worrying about how to translate the value into from human readable string).

Printf() behaviour (buffered print)

b.
- printf buffers the user output until the internal buffer is full before actually output to the screen.
- The trigger indicates the probable size of the internal buffer.
- If a newline character is added, the output is performed "immediately".
- If printf() or similar is used as a debugging mechanism, the buffered output sequence may confuse the coder. e.g. in the original "weird.c", the program can crash without showing any printout, which can easily leads to the wrong conclusion ("the while loop is not executed!").

Fread vs read()

→ fread is a function in stblib C. read is a syscall.

→ fread is buffered. Buffer is shared in process space, not shared between processes

Question on FAT16 vs Ext2

2. **(File System Overhead)** Let us find out the overhead of FAT16 and ext2 file systems. To have a meaningful comparison, we assume that there is a total of 2^{35} 1KB data blocks. Let us find out how much bookkeeping information is needed to manage these data blocks in the two file systems. Express the overhead in terms of number of data blocks.

a. Overhead in FAT16: Give the size of the two copies of file allocation table.
b. Overhead in ext2 is a little more involved. For simplicity, we will ignore the super block and group descriptors overhead.

Below are some known restrictions:
- The two bitmaps (data block and inode) each occupies a single disk block.
- There are **184** inodes per block group.

Calculate the following:
i. Number of data block per block group.
ii. Size of the inode table per block group.
iii. Number of block groups in order to manage 2^{35} data blocks.
iv. Combine (i – iii) to give the total overhead.

c. Comment on the runtime overhead of the two file systems, i.e. how much memory space is needed to support file system operations during runtime.

Important: Size of a page table entry only depends on physical memory space, not virtual memory space, as PTE stores frame number only. Think of it like an array

Ans:

a. Each FAT table is 2^{18} x 16bit (2 bytes) = 2^{37} bytes.
2 copies of FAT table = 2 x 2^{37} = 2^{38} bytes. This takes 2^{38} / 2^{10} = 2^{8} disk blocks.

Note that due to the special codes (free, bad, eof etc), this setup actually handles less than 2^{16} data blocks. We ignore this fact for ease of comparison.

b.
i. Each bitmap is in a 1KB block→ 8K bits → 8K data blocks per block group.
ii. From restrictions, 184 inode x 128 bytes each / 1024 bytes data block = 23 disk blocks.
Note that 184 inodes per block group is an arbitrary number.
iii. From (i) 2^{16} blocks / 2^{13} per block group = 8 block groups.
iv. Overhead per block group = 2 blocks of bitmaps + 23 blocks of inodes = 25 blocks.
Total overhead = 8 x 25 blocks = 200 blocks.

c. FAT: The entire FAT table is in memory, i.e. 2^{37} bytes.
Ext2: Nothing is needed. Though for efficiency, a couple of the recently accessed I-Node may be cached. As a comparison, with the overhead of FAT, we can cache 2^{37}/ 128 bytes = 2^{30} I-node in memory.

Question on Disk IO Scheduling

Questions for your own exploration

5. **(Adapted from AY 2016/17 S 1 exam)**
Most OSes perform some higher-level I/O scheduling on top of just trying to minimize hard disk seeking time. For example, one common hard disk I/O scheduling algorithm is described below:
a. User processes submit file operation requests in the form of **operation(starting hard disk sector, number of bytes)**
b. OS sorts the requests by hard disk sector.
c. The OS merges requests that are nearby into a larger request, e.g. several requests asking for tens of bytes from nearby sectors merged into a request that reads several nearby sectors.
d. The OS then issues the processed requests when the hard disk is ready.

a. How should we decide whether to merge two user requests? Suggest two simple criteria.

b. Give one advantage of the algorithm as described.

c. Give one disadvantage of the algorithm and suggest one way to mitigate the issue.

d. Strangely enough, the OS tends to intentionally delay serving user disk I/O requests. Give one reason why this is actually beneficial using the algorithm in this question for illustration.

e. In modern hard disks, algorithms to minimise disk head seek time (e.g. SCAN variants, FCFS etc.) are built into the hardware controller. i.e. when multiple requests are received by the hard disk hardware controller, the requests will be reordered to minimise seek time. Briefly explain how the high-level I/O scheduling algorithm described in this question may conflict with the hard disk's built-in scheduling algorithm.

Ans:

a. Criterion 1: Requests are in the same or nearby sector (can mention cluster size).
Criterion 2: Requests are of the same type, read / write.
b. Advantage: Seeking latency is reduced.
c. Disadvantage: Potential starvation for user process if the request is not near to existing requests.
Mitigate: Take the request time into account and set certain deadline. Once the deadline is near, issue request regardless of whether it can be merged.
d. Reason to delay: Disk I/O request has very high latency. Delaying the user request for several hundred machine instructions (note that instruction execution time is in the ns range) will not increase the waiting time significantly. However, with more user requests pending, OS can optimize the I/O better. If we do not have enough I/O requests to choose from, merging will not be very effective.
e. Potential conflict: It may turn out that the hard disk controller schedules the requests differently. In the worst case, the scheduling decision by OS may be undone by the controller → time used for sorting / merging are wasted.

Question about relationship between page size and disk block/cluster size

13. [**9 marks**] In virtual memory scheme, some memory pages are stored in a **swap file** on the secondary storage. This question explore the relationship in more details.

a. [**2 marks**] Should the swap file be handled as a normal file? Briefly explain.

b. [**4 marks**] What is the relationship between page size and hard disk cluster size? Briefly explain.

c. [**3 marks**] On most OS, there is only a system-wide swap file shared by all processes. Why do you think the alternative, i.e. per-process swap file, is not a good idea?

13a. [2]	Swap ~~should~~ / should not be handled as a normal file. Reason: Normal file may be spread across different locations on the secondary storage. Paging performance will be affected.
13b. [4]	Relationship: Page size should be the same or multiple of cluster size. Reason: Pages can be efficiently swapped out.
13c. [3]	Reason: As in (a), OS can preallocate a continuous stretch in secondary storage for the system wide swap file. It is also hard to predict the memory usage of a user program.