# Threads

Thread a fundamental unit of cpu utilization.
Parallelism is different from concurrency,
With concurrency it is progress at same time,done by scheduler.
Parallelism needs multiprocessors [runs 2 functions at same time on 2 processors]

Kernels are generally multithread

# Single vs multithreaded process

Single threaded process has code,data,files, register,stack.
In multithreaded, seperate registers,program counter and stack are for each threads,code data, files are shared among threads

# Difference between process and threads

Process is heavy weight, while thread is light weight.

In multicore systems, each core will have seperate timer interrupts.

# User vs Kernel Threads

User threads will give a typedef called threads, the scheduling of threads to be implemented user level library.
Systemcalls->
In many-one model, if one thread does blocking task, all threads are blocked, as process is assumed to block by the kernel

Many-many ,one-one will solve this problem
Tow level model is some are many-many, some one-one

clone-> syscall to create a child process, which takes argument called flags, fork duplicates all, whereas clone can select what exactly you want to clone.
Fork is a wrapper on clone with some flags setup.
clone(files,fs,code,..)

# Thread libraries

Library like pthreads.follow syntax of pthreads in your project to make it POSIX compliant.

# Issues with threads

Does fork() duplicate only calling thread, or all threads?

-> depends on OS
Thread cancellations[]terminating a thread before it has finished]
Cancel immediately
Check periodicallly and cancel itself.

Clone duplicates currently running process, with our requirements,
By default clone() duplicates stack, which duplicates context of process.

For many-one you should not use clone() syscall, using only 1 kernel thread you have to run threads.

For this you need a notion of eip,struct thread to keep a track of all the threads

pthread_create(…,..,fn,….){
t=mallloc(th);
threads[i]=t;
swtch();//library functions available
//similar to xv6's scheduler
}
Struct threads{
Char stack[];
Context c;
Function ptr;
}
First use these libraries and then write those again.
Determine what is context and done.

Library functions->
Getcontext,setcontext, makecontext

Longjump and setjump to change eip values.

## Signals

Synchronous and asynchronous
Signal handler processes signals

signal.c

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

```c
int *p = 1234, i = 1234;
void seghandler(int signo)
{
    printf("seg fault occured \n");
    return;
}
void inthandler(int signo)
{
    printf("INT signal received\n");
    return;
}
int main()
{
    signal(SIGINT, inthandler);
    getchar();
    i = 10;
    signal(SIGSEGV, seghandler);
    *p = 100;
    return 0;
}
```

Ctrl +z ->SIGSTOP
Ctrl +c -> SIGINT
SEGMENT FAULT-> SIGSEV
SIGCONT should make the process continue
int kill(pid_t pid,int sig);

man 7 signals

SIGKILL is number 9
kill -9 <pid>
kill -s SIGINT <pid>
SIGCONT -> continue process after ctrl+z
strace kill -s SIGINT <pid>
strace will show all the syscalls that comand makes

terminal
➜ OS kill -s SIGINT 12569
➜ OS kill -s SIGSTOP 12569
➜ OS kill -9 12569

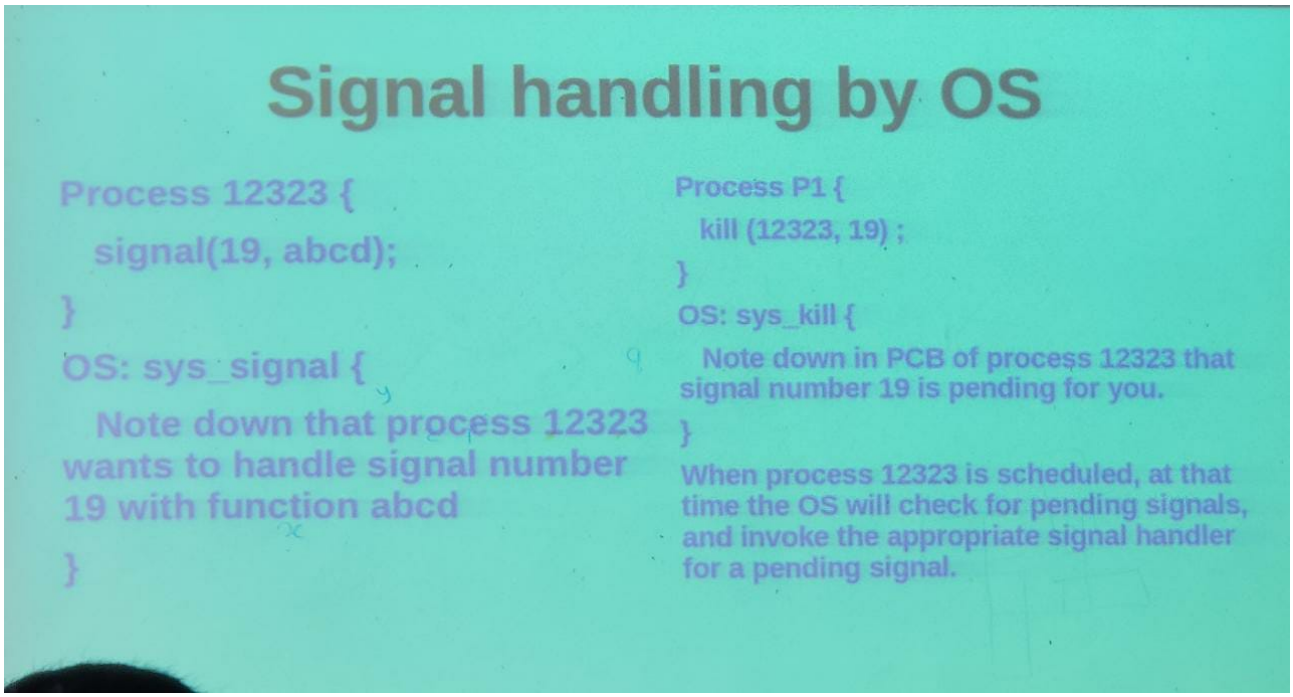
signal() is no longer used, posix uses sigaction()
Signals are identified using numbers too
kill() is used by a process to send another process a signal.
one user cannot send signals to other user's process

man 7 signal
SIGCHILD is sent by kernel to process when child dies

SIGUSR1 is user-defined signal ,send to dd will show statistics

SIGALRM is important for project, it does something like timer

Inside struct proc, the user program that should be run on signal is noted




## Thread pools

Thread in a thread pool will not run function immediately, it is going to wait, the point is chrome know with every thread needs 4 threads,if it didn't creates n number of threads initially, then it may happen enough resource is not available, hence it holds some number of threads when it begins and thus becomes faster, it allows the number of threads in application to be bound to size of pool

## Thread local storage

main(){
th(f)
th(g)
}

Global variables are shared among all threads, which may create problem,
So we will have a data global to all functions of a thread

[global is accessible in all files, global static is accessed inside the same file only]
Solved by thread-local storage

## Scheduler activations for threads

Suppose all the kernel threads in the many-many mapping gets blocked, then how will the application know that the kernel threads are blocked, so kernel has to tell the application, so some communication from kernel to user is required[backward communication]
For this we have scheduler activations
It provides upcalls which is a backward communication mechanism from kernel to thread library

So between user thread and kernel thread is a lightweight process,
How does this solve?
LWP is a part of kernel data structure,

In linux there is no difference between thread and process, there is only 1 term called task

See fork's syscall using strace

Kinit1, kinit2 -> free list
Kalloc,kfree ->managing free frame list
setupkvm()
Kpgdir-> entries of pgdir point to page tables,
Can we move seginit to line 35?

userinit() -> creation of user processes
userinit() creates init process, it is an application process, we have to set it up without using fork,

_binary_initcode_start[] is a starting address of 'start' symbol in binary file called initcode, similarly _binary_initcode_size[] is the size (2c)
It can be searched in kernel.sym

```
kernel: $(OBJS) entry.o entryother initcode kernel.ld
	$(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b
binary initcode entryother
```

-b means include symbols from these files
Essentially initcode.S is the initcode file.

We are creating a process called initcode, whose only job is to exec init

userinit calls allocproc()

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
// UNUSED=0
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
```

```
  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;

  return p;
}
```

nextpid is global variable initialized to 1.
Only way a process is created is using trap

The data structure used in kalloc() and kfree() in xv6 is
. Singly linked NULL terminated list

In xv6, The struct context is given as
struct context {
  uint edi;
  uint esi;
  uint ebx;
uint ebp;
uint eip; };
Select all the reasons that explain why only these 5 registers are included in the struct context.
->The segment registers are same across all contexts, hence they need not be saved, eax, ecx, edx are caller save, hence no need to save, esp is not saved in context, because context{} is on stack and it's address is always argument to swtch()

Hierarchical Paging → More memory access time per hierarchy, Inverted Page table → Linear/Parallel search using page number in page table, Hashed page table → Linear search on collsion done by OS (e.g. SPARC Solaris) typically

the status of segmentation setup in xv6
entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor

Page fault handling
Process is kept in wait state → 6, The reference bit is found to be invalid by MMU → 1, Restart the instruction that caused the page fault → 9, Disk interrupt wakes up the process → 7, A hardware interrupt is issued → 2, Page tables are updated for the process → 8, OS makes available an empty frame → 4, OS schedules a disk read for the page (from backing store) → 5, Operating system decides that the page was not in memory → 3

correct statements about sched() and scheduler() in xv6 code
sched() and scheduler() are co-routines, When either sched() or scheduler() is called, it does not return immediately to caller, When either sched() or scheduler() is called, it results in a context switch, sched() switches to the scheduler's context, scheduler() switches to the selected process's context, After call to swtch() in scheduler(), the control moves to code in sched(), After call to swtch() in sched(), the control moves to code in scheduler(), Each call to sched() or scheduler() involves change of one stack inside swtch()

return a free page, if available; 0, otherwise → kalloc(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Setup kernel part of a page table, and switch to that page table → kvmalloc(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[]

paging with demand paging and select the correct s
Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False
The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True
The stack allocated in entry.S is used as stack for scheduler's context for first processor: True
The kernel code and data take up less than 2 MB space: True
The free page-frame are created out of nearly 222 MB: True
The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True

xv6 uses physical memory upto 224 MB only: True
The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False
The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False
The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True PHYSTOP can be increased to some extent, simply by editing memlayout.h: True
Cumulative size of all programs can be larger than physical memory size: True One Program's size can be larger than physical memory size: True
Logical address space could be larger than physical address space: True Relatively less I/O may be possible during process execution: True
Virtual access to memory is granted: False Virtual addresses are available: False
Code need not be completely in memory: True

the actions done as part of code of swtch() in swtch.S
Switch from old process context to new process context: False
Restore new callee saved registers from user stack of new context: False Jump to code in new context: False
Save old callee saved registers on kernel stack of old context: True
Switch from one stack (old) to another(new): True
Save old callee saved registers on user stack of old context: False
Restore new callee saved registers from kernel stack of new context: True

which explains how paging works.
The locating of the page table using PTBR also involves paging translation: False
Size of page table is always determined by the size of RAM: False
The physical address may not be of the same size (in bits) as the logical address: True The page table is indexed using frame number: False
The page table is itself present in Physical memory: True
The page table is indexed using page number: True
The PTBR is present in the CPU as a register: True
Maximum Size of page table is determined by number of bits used for page number: True

statements about MMU and it's functionality

MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

the compiler's view of the process's address space, for each of the following MMU schemes:
Segmentation, then paging → many continuous chunks of variable size, Paging → one continuous chunk, Segmentation → many continuous chunks of variable size, Relocation + Limit → one continuous chunk

a processor supports base(relocation register) + limit scheme of MMU. The process sets up it's own relocation and limit registers when the process is scheduled: False
The OS sets up the relocation and limit registers when the process is scheduled: True
The compiler generates machine code assuming appropriately sized semgments for code, data and stack.: False
The hardware may terminate the process while handling the interrupt of memory violation: False
The OS may terminate the process while handling the interrupt of memory violation: True
The hardware detects any memory access beyond the limit value and raises an interrupt: True
The OS detects any memory access beyond the limit value and raises an interrupt: False
The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;: True

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived,are: end, P2V(PHYSTOP)

Order events in xv6 timer interrupt code
alltraps() will call iret → 18, Process P2 is executing → 14, change to context of the scheduler, scheduler's stack in use now → 11, trap() is called → 7, change to context of P2, P2's kernel stack in use now → 13, P2's trap() will return to alltraps → 17, P2's yield() will return in trap() → 16, Change of stack from user stack to kernel stack of P1 → 3, Trapframe is built on kernel stack of P1 → 6, jump to alltraps → 5, P1 is marked as RUNNABLE → 9, jump in vector.S → 4, P2 will return from

sched() in yield() → 15, P2 is selected and marked RUNNING → 12, Timer interrupt occurs → 2, Process P1 is executing → 1, sched() is called, → 10, yield() is called → 8