



POLITECNICO DI TORINO

# Analisi comparativa tra Os161 e Xv6

Progetto di Programmazione di Sistema - AA 23/24

---

PRESENTATO DA:

Nunzio Messineo  
Gabriele Martina

CONTATTO:

[s315067@studenti.polito.it](mailto:s315067@studenti.polito.it)  
[s310789@studenti.polito.it](mailto:s310789@studenti.polito.it)

# Introduzione

## Xv6

---

Sistema operativo con scopi didattici  
basato su UNIX V6, progettato dal MIT  
nel 2006

Realizzato per lo più in C a Assembly

Progettato per essere eseguibile su  
architetture x86 nella versione del 2006  
e RISC-V nel 2020

Architettura in studio: x86

## Os161

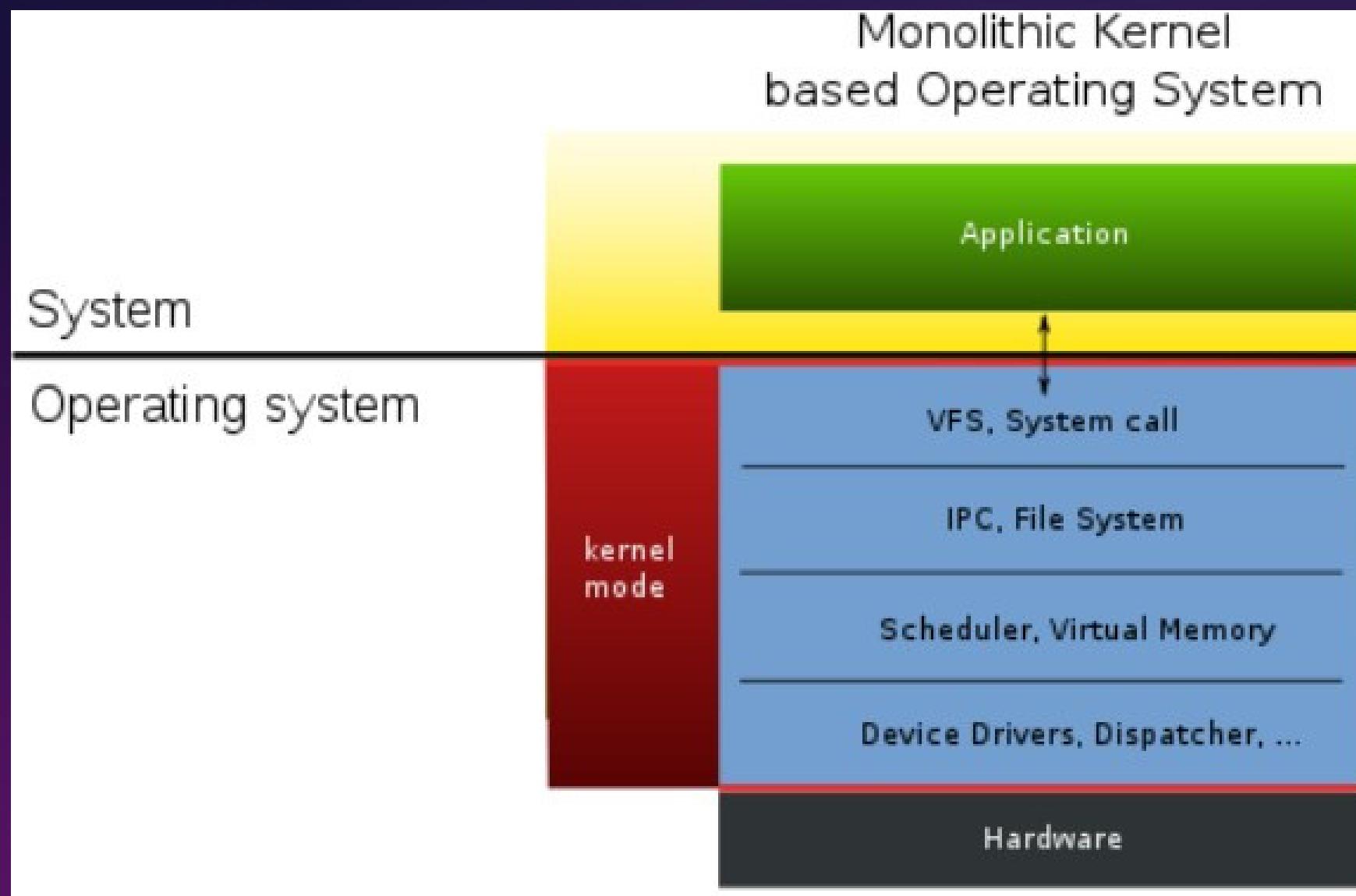
---

Os161 è progettato per l'istruzione e sviluppato  
presso l'Università di Harvard

Realizzato per lo più in C a Assembly

Supporta diverse architetture, inclusi MIPS,  
ARM e Intel

Architettura in studio: MIPS

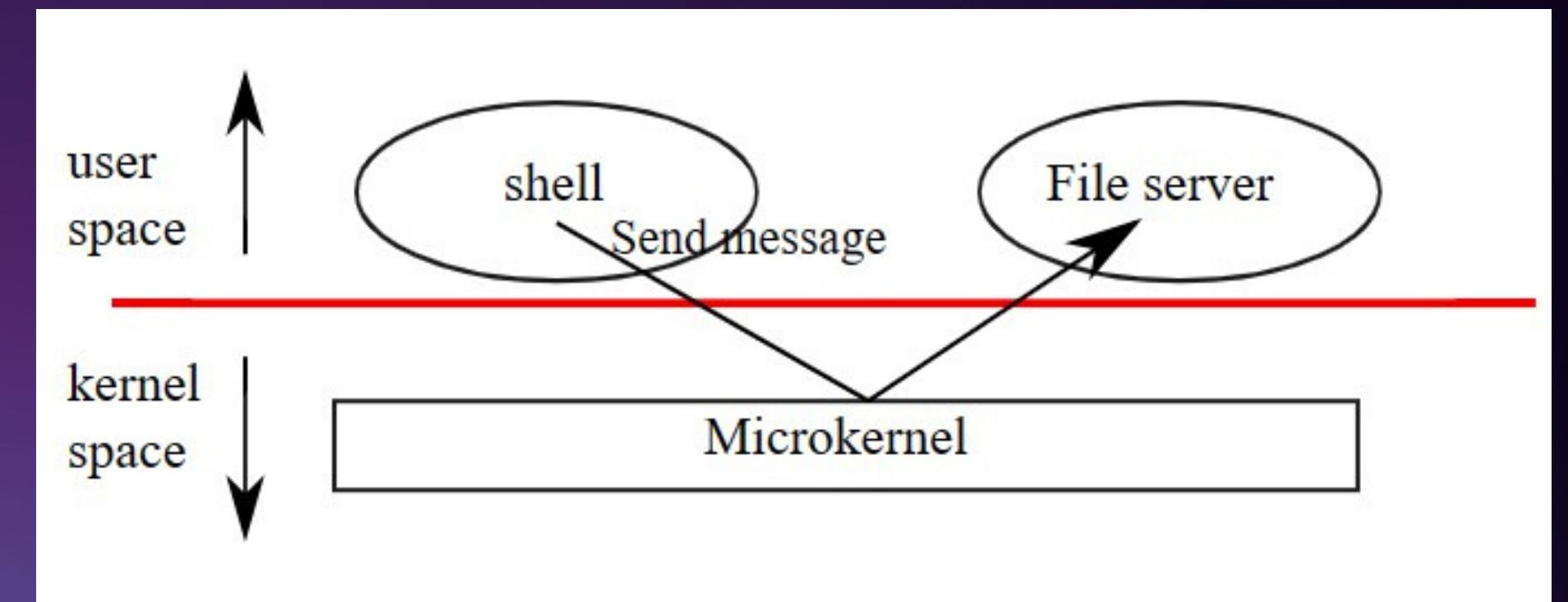


# Struttura del Kernel

Il sistema operativo ha una struttura **monolitica**. L'intero sistema operativo viene eseguito con pieno privilegio hardware. Poiché Xv6 non fornisce molti servizi, il suo kernel è più piccolo di alcuni microkernel.

# Struttura del Kernel di Os161

Utilizza un'organizzazione a microkernel, dove solo le funzioni essenziali risiedono nel kernel, mentre i servizi aggiuntivi sono implementati come server a livello utente.



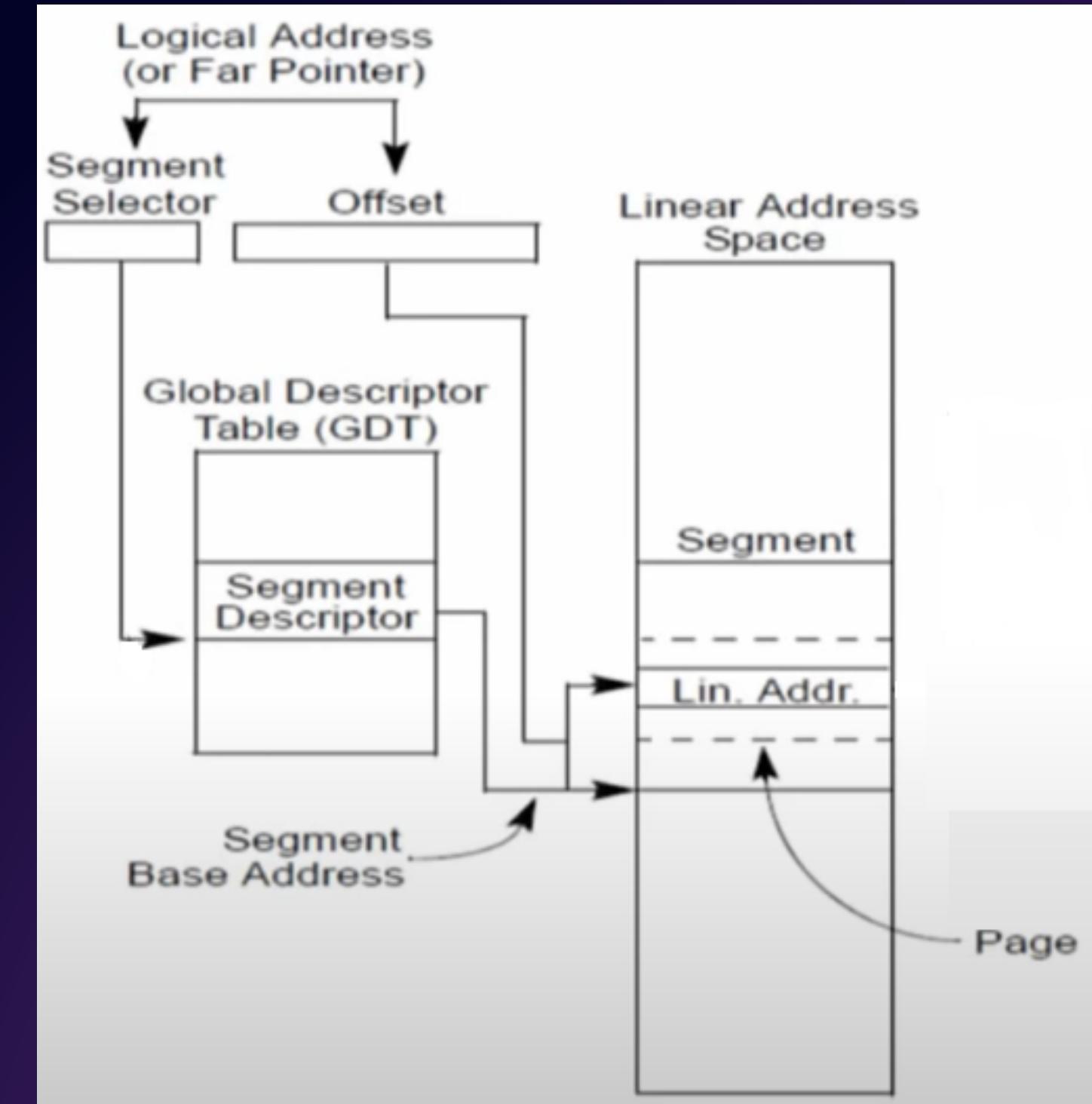
# Gestione della memoria

---

Implementa un sistema di gestione della memoria virtuale basato su **paging**.

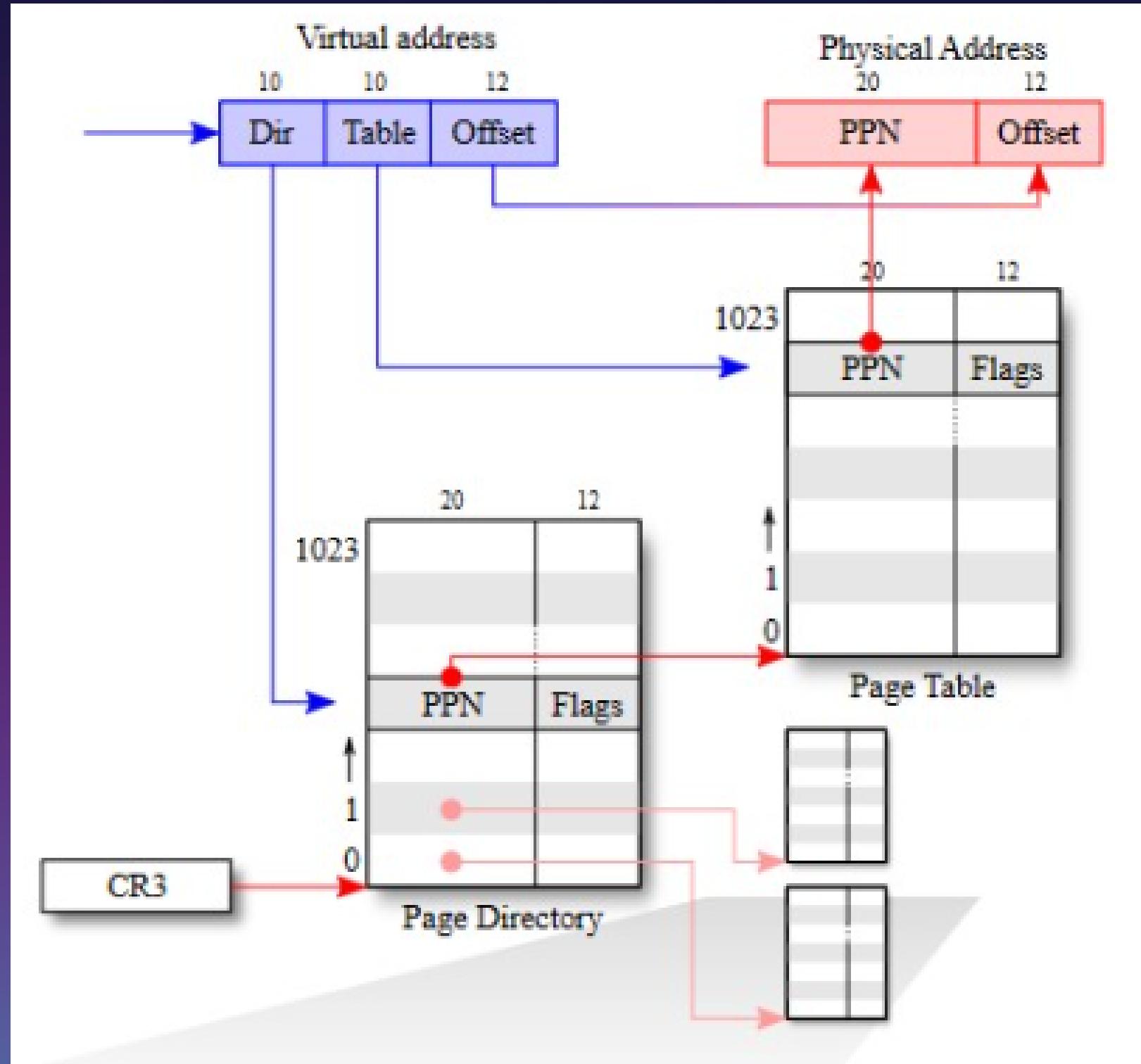
Il paging è diviso in due fasi:

- Fase di **segmentazione** in cui si ottiene 'l'indirizzo lineare' partendo dall'indirizzo virtuale.



# Gestione della memoria

- Le **page tables** consentono il controllo degli indirizzi di memoria, permettendo a Xv6 di multiplexare gli spazi degli indirizzi di processi diversi su una singola memoria fisica e proteggere le memorie di processi diversi.



# Gestione della memoria

Os161

## Analogie

---

Come in xv6 adotta la virtual memory  
tramite l'utilizzo di page table

## Differenze

---

Os161 implementa varie tecniche avanzate per ottimizzare l'utilizzo della memoria, tra cui l'utilizzo di memoria condivisa (shared memory), il demand paging e la condivisione in lettura/scrittura (Copy-On-Write, COW).

# System Call, Exceptions e Interrupts

In Xv6 vengono tutte gestite da una stessa funzione. Come prima cosa bisogna effettuare il context switch per passare da user mode a kernel mode. Nello specifico il sistema operativo deve:

1. Salvare i registri del processore per un futuro ripristino trasparente.
2. Essere configurato per l'esecuzione nel kernel scegliendo un punto in cui far iniziare l'esecuzione del kernel.
3. Il kernel deve essere in grado di recuperare informazioni sull'evento, ad esempio gli argomenti della chiamata di sistema.

# Gestori di trap in Xv6

- La gestione delle trap viene gestita a livello assembly tramite la funzione **int n** (in usys.S) (dove nel caso delle system call n = T\_SYSCALL)
- Successivamente avviene la creazione del trap frame con il salvataggio dei registri nella funzione **alltraps**
- int n infine chiama trap a livello c che determina cosa ha scatenato l'interrupt
- Per le system call invoca **syscall** in (syscall.c) che chiama la funzione corrispondente all'ID (in %eax)
- **trapret** (in trapasm.S) esegue il context switch inverso e conclude la chiamata

# Creazione di una system call

L'aggiunta di una nuova system call a Xv6 richiede la modifica del codice del kernel e l'aggiunta di una nuova voce alla system call table:

1. Definire una nuova system call nel codice del kernel

```
#define SYS_getyear 22
```

2. Aggiungerla alla system call table (in syscall.h) con un codice univoco

```
[SYS_getyear] sys_getyear
```

3. Aggiungere un puntatore alla chiamata di sistema (in syscall.c solo il prototipo)

```
extern int sys_getyear(void)
```

4. Aggiungere l'interfaccia utente in usys.S

```
SYSCALL(getyear)
```

5. Implementare la system call function in sysproc.c

```
int sys_getyear(void) { return 1975; }
```

6. Modificare il Makefile

```
make clean; make
```

# Gestione delle trap in Os161

In Os161, allo stesso modo di Xv6, ha un solo handler per gestire system call, exception e interrupt. Le principali differenze sono:

- *Differenze di carattere implementativo*: dovute principalmente all'architettura (MIPS vs x86)
- *Astrazione e modularità*: OS161 promuove una maggiore astrazione e modularità rispetto a xv6. Le system call sono implementate come chiamate remote attraverso messaggi, il che consente una maggiore separazione tra il codice utente e il codice kernel.
- *Isolamento e protezione*: Grazie alla separazione dei servizi e dei componenti nel microkernel di OS161, è più probabile che il sistema operativo offra un maggiore isolamento e protezione tra le diverse componenti. Questo è vantaggioso in termini di sicurezza e stabilità

Anche l'implementazione delle system call hanno sostanzialmente lo stesso funzionamento.

# Sincronizzazione in Xv6

I meccanismi di sincronizzazione implementati da Xv6 sono:

- ***Spin Lock***: è un tipo di lock caratterizzato principalmente da un booleano che determina se è detenuto o è disponibile.
- ***Sleep Lock***: è un tipo di lock che può essere rilasciato e acquisito da un thread diverso da quello che lo detiene.
- ***Wakeup*** e ***Sleep***: Sleep e Wakeup consentono ai processi nello stato di sleeping di dormire in attesa di un evento e ad un altro processo di svegliarlo una volta che l'evento è avvenuto.

# Sincronizzazione in Os161

Os161, allo stesso modo di Xv6, implementa spinlock, mutex lock (sleep lock) e wait channel (wakeup e sleep) però aggiunge una tecnica di sincronizzazione più sofisticata che è quella dei semafori.

Inoltre a differenza di xv6, in Os161 i processi hanno un campo extra che ne definisce la priorità in modo che i thread con priorità più bassa non possono interrompere l'esecuzione di un thread con priorità superiore.

# Scheduling - Xv6

Xv6 implementa un semplice algoritmo di scheduling a **round-robin** che assegna a ciascun processo un intervallo di tempo fisso, chiamato **quantum**. È un algoritmo a priorità fissa in quanto tutti i processi hanno la stessa importanza nell'essere eseguiti.

Il multiplexing dei processi avviene usando il meccanismo di sleep e wakeup. Lo scheduling è costituito principalmente dalle seguenti fasi:

- Acquisizione del lock ptable.lock
- Context switch con l'uso della funzione swtch
- Algoritmo di scheduling
- Ritorno al contesto precedente
- Rilascio del lock ptable.lock

# Scheduling - Os161

Analogamente a Xv6, Os161 implementa di default utilizza una coda unica round-robin.

Come principale differenza in Os161 sono già definite le priorità in modo da assegnare ad ogni thread un'importanza diversa a seconda delle funzionalità o del contesto.

# File System - Xv6

Il file system di Xv6 è suddiviso nei seguenti layer:

- Il livello **Disk**: legge e scrive blocchi su un hard disk IDE.
- Il livello **Buffer cache**: ha due funzioni fondamentali: sincronizzare l'accesso ai blocchi del disco e memorizzare nella cache i blocchi più popolari
- Il livello **Logging**: consente ai livelli superiori di incapsulare gli aggiornamenti a diversi blocchi in una transazione e garantisce che i blocchi vengano aggiornati atomicamente in caso di arresti anomali.
- Il livello **Inode** e **Block Allocator**: fornisce file senza nome, ciascuno rappresentato utilizzando un inode e una sequenza di blocchi che contengono i dati del file.

# File System - Os161

Il file system di Os161 è chiamato Simple File System (SFS) e possiede al suo interno buona parte delle funzionalità di base di un fs:

- Directory gerarchiche
- Supporto per file di grandi dimensioni tramite blocchi indiretti multilivello
- Blocco a grana fine
- Un cache buffer di dati e metadati del file system

## Sezione 2

# Implementazione nuove funzionalità in xv6

System call  
Gestione di priorità dei processi

# Esecuzione di xv6 su Ambiente Linux

## 1. Scarica e Compila:

- Clona il repository di xv6 con **git clone**.
- Aggiorna i permessi di xv6: **chmod 700 -R xv6-public**
- Installa gli strumenti essenziali con: **sudo apt-get install build-essential qemu-system-x86**.
- Compila con **make clean** e **make**.

## 2. Esegui con QEMU (emulatore):

- Avvia con **make qemu**.



# Debug in xv6

## Shell #1

- cd xv6-public
- make qemu-gdb

```
gab@gab-VirtualBox:~/xv6-public$ make qemu-gdb
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o ptime.o ptime.c
ld -m elf_i386 -N -e main -Ttext 0 -o _ptime ptime.o ulib.o usys.o printf.o umalloc.o
objdump -S _ptime > ptime.asm
objdump -t _ptime | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > ptime.sym
./mkfs fs.img README.oldname.txt _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests _wc _zombie
_shutdown _chpriority _getppid _dproc _pwd _ptime
nmeta 59 (boot, super, log blocks 30 inode blocks 26, bitmap blocks 1) blocks 941 total 1000
balloc: first 857 blocks have been allocated
balloc: write bitmap block at sector 58
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tpl > .gdbinit
*** Now run 'gdb'.
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=qcow2 -smp 2 -m 512 -S -gdb tcp::26000
```

## Shell #2

- gdb kernel
- target remote: TCP-PORT

# System call implementate

- **sys\_shutdown** Consente l'uscita da QEMU e da xv6
- **sys\_cps** Restituisce un quadro generale dei processi attivi in xv6
- **sys\_getppid** Restituisce il **pID** del processo padre del processo corrente
- **sys\_chpriority** Permette di cambiare la **priorità** di un processo
- **sys\_getcwd** Restituisce il **path** della *work directory* corrente

# Nuove funzioni implementate

- **dProc**
- **is\_ancestor**

Avvia un nuovo processo e ne genera un numero definito di processi *child*.

Tutti con priorità di default 10

Dati due processi qualsiasi p1 e p2, mi dice se p2 è *antenato* di p1

```
#define SYS_cps    22
#define SYS_shutdown 23
#define SYS_chpriority 24
#define SYS_getppid 25
#define SYS_getcwd 26
```

*syscall.h*

```
// Function to check if p1 is an ancestor of p2
int is_ancestor(struct proc *p1, struct proc *p2) {
    while (p2 != NULL) {
        if (p2 == p1) {
            return 1; // p1 is an ancestor of p2
        }
        p2 = p2->parent;
    }
    return 0; // p1 is not an ancestor of p2
}
```

*proc.c*

```
int main(int argc, char *argv[]) {
    int num_children;
    int dot;
    // Imposta il numero di figli
    if (argc < 2)
        num_children = 1; // Default
    else
        num_children = atoi(argv[1]);
    if (num_children < 0 || num_children > 20)
        num_children = 2;
    for (int i = 0; i < num_children; i++) {
        int pid = fork();

        if (pid < 0) {
            printf(1, "%d failed in fork\n", getpid());
        } else if (pid > 0) {
            // Codice del genitore
            printf(1, "Parent %d creating child %d\n", getpid(), pid);
            wait();
        } else {
            // Codice del figlio
            printf(1, "Child %d created\n", getpid());

            // Fa passare del tempo rendendo attivo il processo
            for (int z = 0; z < 10000000; z++) { //fa operazioni complesse
                dot += 800000.73*3.14;
                //sleep(10); // Introduce un ritardo di 10 ms
            }
            break;
        }
    }
    // L'exit termina il processo figlio
    exit();
}
```

*dproc.c*

# sys\_shutdown

L'operazione di ***shutdown*** di un sistema operativo è estremamente delicata

Questa dovrebbe prevedere controlli su: file system, sincronizzazione  
processi, ecc...

Una system call non banale per un sistema operativo complesso.

xv6 nasce a scopi didattici e per questo non è necessario un controllo accurato e potenzialmente un' uscita forzata, senza controlli, può non portare ad errori o perdita di dati.

# sys\_shutdown

Il nostro progetto prevede due varianti di **shutdown**.

- **Forzato**: nessun controllo. Equivale allo spegnimento improvviso di un computer o dell'emulatore.
- **Rispettando lo scheduling dei processi**: si segue la priorità dei processi (*dal meno importante al più importante*) facendo attenzione a non terminare subito: il processo corrente o un suo antenato (is\_ancestor).

```
int shutdown (int choice) {  
    se choice == 0:  
        outw(); //manda il comando a QEMU per spegnimento  
    altrimenti:  
        pshutdown();  
}
```

```

int shutdown(int choice){
    if (choice == 1){
        pshutdown();
        return 0;
    }
    cprintf("Spegimento forzato\n");
    outw(0xB004, 0x0|0x2000);
    outw(0x604, 0x0|0x2000); // Send the shutdown command
    return 0; // This function should not return;
}

```

*proc.c*

```

void pshutdown(void){
    struct proc *p;

    // Sort processes based on priority in ascending order
    for (int i = 0; i < NPROC; i++) {
        for (int j = 0; j < NPROC - 1; j++) {
            // Compare priorities and swap if needed
            if (ptable.proc[j].priority > ptable.proc[j + 1].priority) {
                struct proc temp = ptable.proc[j];
                ptable.proc[j] = ptable.proc[j + 1];
                ptable.proc[j + 1] = temp;
            }
        }
    }

    // Close processes with lower priority first (excluding the current process and those with priority <= 5)
    for (p=ptable.proc; p<&ptable.proc[NPROC]; p++) {
        if (p->state != UNUSED && p->state != ZOMBIE && p->priority > 5 && p != myproc()) {
            // Check if the process is not an ancestor of the current process
            if (is_ancestor(myproc(), p) == 0) {
                // Terminate the process
                kill(p->pid);
            }
        }
    }

    // Call the shutdown function after killing processes
    cprintf("Spegimento strutturato processi\n");
    outw(0xB004, 0x0|0x2000);
    outw(0x604, 0x0|0x2000); // Send the shutdown command
    // Close the current process (the one that called the system call)
    exit();
}

```

*proc.c*

# sys\_getppid

```
int  
sys_getppid(void)  
{  
    return myproc()->parent->pid;  
}
```

*sysproc.c*

# sys\_cps

Permette di avere un quadro completo di tutti i processi salvati sul momento, a prescindere dal loro stato, all'interno della ***ptable*** (Page Table).

```
int cps(void) {  
    sti(); // set interrupt enable  
    lock(ptable);  
    for(ptable->p){  
        print(p->...);  
    }  
    unlock(ptable)  
}
```

# sys\_cps

```
int
cps(void)
{
    struct proc *p;
    sti();
    acquire(&ptable.lock);

    cprintf("NAME \t PID \t STATE \t PRIORITY\n");
    for (p=ptable.proc; p<&ptable.proc[NPROC]; p++){
        if(p->state == RUNNING)
            cprintf("%s \t %d \t RUNNING \t %d \n", p->name, p->pid, p->priority);
        else if(p->state == SLEEPING)
            cprintf("%s \t %d \t SLEEPING \t %d \n", p->name, p->pid, p->priority);
        else if(p->state == RUNNABLE)
            cprintf("%s \t %d \t RUNNABLE \t %d \n", p->name, p->pid, p->priority);
        else if(p->state == UNUSED)
            cprintf("%s \t %d \t UNUSED \t %d \n", p->name, p->pid, p->priority);
        else if(p->state == EMBRYO)
            cprintf("%s \t %d \t EMBRYO \t %d \n", p->name, p->pid, p->priority);
        else if(p->state == ZOMBIE)
            cprintf("%s \t %d \t ZOMBIE \t %d \n", p->name, p->pid, p->priority);
        else if(p == NULL)
            break;
    }
    release(&ptable.lock);
    return 0;
}
```

proc.c

# sys\_chpriority

**xv6** è un sistema operativo semplice e, di conseguenza, lo sono anche i suoi *meccanismi di scheduling dei processi*.

Nativamente, lo scheduling avviene per mezzo di un semplice **algoritmo di ROUD ROBIN** e *il concetto di priorità non esiste*.

*Sys\_chpriority* nasce per introdurre una gestione dei processi più complessa, attribuendo ad ognuno una **priorità**.

*Più sarà alta (20) e meno importante sarà il processo, e viceversa(0). La priorità di default è 10.*

Quando un processo viene creato, verrà schedulato per priorità crescenti (dal più importante al meno), in modo da ridurre i tempi di attesa per tutti quei processi che devono svolgere ruoli più significativi nel sistema operativo.

# sys\_chpriority

1. Aggiunto il campo *priority* nella struttura del processo

```
struct proc {  
    uint sz;  
    pde_t* pgdir;  
    char *kstack;  
    enum procstate state;  
    int pid;  
    struct proc *parent;  
    struct trapframe *tf;  
    struct context *context;  
    void *chan;  
    int killed;  
    struct file *ofile[NOFILE];  
    struct inode *cwd;  
    char name[16];  
    //**NEW**  
    int priority;  
};
```

2. Nelle funzioni *allocproc()*, *wait()*, in *proc.c* assegno la priorità di default 10

```
p->priority = 10;
```

3. Aggiorno lo scheduling in *scheduler()* (*proc.c*)

```
***NEW***  
struct proc *bestPriority;  
  
Loop over process table looking for process to run.  
Acquire(&ptable.lock);  
(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  
    if(p->state != RUNNABLE)  
        continue;  
  
    ***NEW***  
    bestPriority = p;  
    //choose one with highest priority  
    for (p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){  
        if(p1->state != RUNNABLE)  
            continue;  
        if(bestPriority->priority > p1->priority) //larger value, lower priority  
            bestPriority = p1;  
  
= bestPriority;  
  
// Switch to chosen process. It is the process's job  
// to release ptable.lock and then reacquire it  
// before jumping back to us.  
p->proc = p;  
switchuvm(p);
```

# sys\_chpriority

## 4. Implemento la system call.

```
int sys_chpriority(void){  
    int pid, pr;  
    if(argint(0, &pid)<0)  
        return -1;  
    if(argint(1, &pr)<0)  
        return -1;  
    return chpriority(pid, pr);  
}
```

*sysproc.c*

```
int chpriority(int pid, int priority) {  
    struct proc *p;  
    acquire(&ptable.lock);  
  
    for(p=ptable.proc; p<&ptable.proc[NPROC]; p++){  
        if(p->pid == pid){  
            p->priority = priority;  
            break;  
        }  
    }  
    release(&ptable.lock);  
    return pid;  
}
```

*proc.c*

# sys\_chpriority

## 5. Implemento una funzione di test per permettere l'interazione utente/sistema operativo

```
int main(int argc, char *argv[]) {
    int target_pid, new_priority;

    if (argc < 3) {
        printf(2, "Usage: setpriority <pid> <priority>\n");
        exit();
    }

    target_pid = atoi(argv[1]);
    new_priority = atoi(argv[2]);

    if (new_priority < 0 || new_priority > 20) {
        printf(2, "Invalid priority (0-20)!\n");
        exit();
    }

    printf(1, "Setting priority of process %d to %d\n", target_pid, new_priority);
    if (chpriority(target_pid, new_priority) != target_pid)
        printf(1, "Error!\n");

    exit();
}
```

*chpriority.c*

# sys\_getcwd

La funzione **getcwd** è utilizzata per ottenere il percorso del directory di lavoro corrente (current working directory, CWD) di un processo.

# sys\_getcwd

1. Passiamo 3 variabili (di cui due prese dallo stack):

- **n**: la lunghezza del buffer in cui riportare il path
- **p**: il buffer
- **myproc()->cwd**: la directory corrente

```
int  
sys_getcwd(void)  
{  
    char *p;  
    int n;  
    if (argint(1, &n) < 0 || argptr(0, &p, n) < 0){  
        return -1;  
    }  
    return name_for_inode(p, n, myproc()->cwd);  
}
```

# sys\_getcwd

Gli **inode** sono al di sotto della gerarchia del filesystem e dei nomi dei file (*file.h*). La gerarchia è creata con inode speciali di tipo TDIR, che contengono strutture **dirent** (tuple di: un nome e inode del file corrispondente proprio a quel nome). Gli **inode** non contengono direttamente il concetto di nome del file e possono esistere in diverse directory con nomi di file diversi.

Il sistema operativo non converte direttamente gli *inode* in nomi di file.

*proc->cwd* è sempre una directory e per fortuna la funzione *create* crea l'entry point .. che punta sempre a una directory *parent*.

Il tutto si riassume con una funzione ricorsiva

# sys\_getcwd

2.

```
func psuedocode_func(inode):
    if inode is root:
        return "/"
    else:
        name := ""
        for dirent in inode.parent:
            if dirent.inode == inode:
                name = dirent.name
                break
        if not name:
            panic("inode not in parent")
        return psuedocode_func(inode.parent) + name + "/"
```

```
int
name_for_inode(char* buf, int n, struct inode *ip) {
    int path_offset;
    struct inode* parent;
    char node_name[DIRSIZ];
    if( ip->inum == namei("/")->inum) { //if inode is the root
        buf[0] = '/';
        return 1;
    } else if (ip->type == T_DIR) { // types of inode are in stat.h
        parent = dirlookup(ip, "..", 0);
        ilock(parent); // inode upload from disk
        if (name_of_inode(ip, parent, node_name)) {
            cprintf("Error: not find name of inode");
        }
        path_offset = name_for_inode(buf, n, parent);
        safestrcpy(buf + path_offset, node_name, n-path_offset);
        path_offset += strlen(node_name);
        if (path_offset == n-1) {
            buf[path_offset] = '\0';
            return n;
        } else {
            buf [path_offset++] = '/';
        }
        iunlock(parent); //free
        return path_offset;
    } else if(ip->type == T_DEV || ip -> type == T_FILE) {
        cprintf("Process cwd is not a directory");
    } else {
        cprintf("unknown inode type");
    }
    return -1;
}
```

sysfile.c

# sys\_getcwd

3. **Dirent** è solo una struttura di un *ID inode ushort e un nome*. Ciò significa che tutto ciò che fa questo ciclo è eseguire il loop su ogni voce di directory nella directory, caricando la **de** con la funzione **readi**.

Controlliamo **de** per vedere se si riferisce al nostro *inode* ricercato, **ip**. Se lo è, usiamo **safestrcpy** per copiarlo nel buffer ed eseguire un ritorno con successo

```
int
name_of_inode(struct inode *ip, struct inode *parent, char buf[DIRSIZ]){
    uint off;
    struct dirent de;
    /*
    In xv6 directories are a list of dirent structures (made up name and id)
    |This for-cycle read every item in the list and upload them in de
    */
    for (off = 0; off < parent->size; off += sizeof(de)) {
        if(readi(parent, (char*)&de, off, sizeof(de)) != sizeof(de)){
            cprintf("Error: read dir entry");
        }
        // check if de == our inode
        if(de.inum == ip->inum) {
            safestrcpy(buf, de.name, DIRSIZ);
            return 0;
        }
    }
    return -1;
}
```

sysfile.c

# Come aggiungere nel Makefile le funzioni implementate

1.

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _cps\
    _shutdown\
    _chpriority\
    _getppid\
    _dproc\
    _pwd\
```

2.

```
EXTRA=\
    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
    printf.c umalloc.c cps.c shutdown.c chpriority.c getppid.c dproc.c pwd.c\
    README oldname.txt dot-bochssrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tpl gdbutil\
```

# Risultati

```
cps          2 19 14252
shutdown    2 20 14404
chpriority  2 21 14872
getppid     2 22 14204
dproc       2 23 15376
pwd          2 24 14436
console     3 25 0

$ getppid
PID of my parent: 2
$ pwd
/
$ cps
NAME      PID STATE   PRIORITY
init      1 SLEEPING 5
sh        2 SLEEPING 5
cps       6 RUNNING  5
0         0 UNUSED   0
0         0 UNUSED   0
0         0 UNUSED   0
```

- **ls** permette di vedere tutte le funzioni implementate e usabili nel sistema operativo
- **pwd** (implementa sys\_getcwd) restituisce solo '**/**' in quanto ci troviamo nella *radice*
- **cps** stampa tutti i processi. I processi di inizializzazione di sistema hanno sempre priorità più alta di quella di default (*exec.c*). Lo stesso vale per il processo *cps* che è figlio di *sh* (notare *getppid*)

# Risultati

```
$ dproc &; dproc&
$ Parent 6 creating child 8
Child 8 created
Parent 7 creating child 9
Child 9 created
cps
```

NAME	PID	STATE	PRIORITY
init	1	SLEEPING	5
sh	2	SLEEPING	5
dproc	8	RUNNABLE	10
dproc	7	SLEEPING	5
dproc	6	SLEEPING	5
dproc	9	RUNNABLE	10
cps	10	RUNNING	5
	0	UNUSED	0
	0	UNUSED	0

- *dproc* in questo caso è stato chiamato 2 volte: crea due processi (pid 6 e 7) e da essi genera rispettivamente un processo child (pid 8 e 9)

# Risultati

```
$ dproc &
$ Parent 4 creating child 5
Child 5 created
$ cps
exec: fail
exec $ failed
$ cps
NAME      PID      STATE      PRIORITY
init      1        SLEEPING    5
sh        2        SLEEPING    5
dproc     5        RUNNABLE   10
dproc     4        SLEEPING    5
cps       7        RUNNING    5
exit status 0
$ chpriority 5 16
Setting priority of process 5 to 16
$ cps
NAME      PID      STATE      PRIORITY
init      1        SLEEPING    5
sh        2        SLEEPING    5
dproc     5        RUNNABLE   16
dproc     4        SLEEPING    5
cps       9        RUNNING    5
exit status 0
^
```

**FINE**