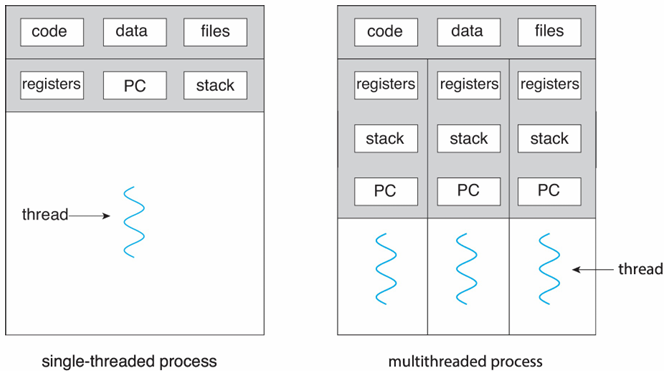
# OS161 OVERVIEW

**THREAD**: stato di controllo di un programma in esecuzione con dei dati veri.

* Caratterizzato da **contesto/context**: ciò che viene salvato e ripristinato in caso di cambi.
  + Stato CPU: *PC, staco pointer, registri, modo di esecuzione privilegiato/non*
  + Stack, localizzato nello spazio di indirizzamento del processo
* Memoria:
  + Codice programma
  + Dati programma
  + Stack programma, contenenete registrazioni delle procedure di attivazione.

#### **Immagine che contiene testo, schermata, Carattere, linea Descrizione generata automaticamenteThread si dividono in:**

* **Thread-user**
  + Necessitano di librerie con funzioni per creare e gestire thread.
* **Thread-kernel** (noi vediamo solo questi)
  + Girano con modalità privilegiata in spazio di memoria virtuale user

**Processo**: programma in esecuzione che comprente:

* Codice programma → text section
* Program counter
* Stack
* Data section
* Heap (nota: in windows c’è anche la possibilità di separare)

I thread in un processo condividono: codice, data e files; il resto è duplicato.

*Nota: In contesti lavorativi si preferiscono multi-processi e non multi-thread. Multi-thread è più efficiente dal punto di vista della concorrenza ma meno usato in quanto: se in un processo multi-thread, si pianta un thread, si blocca tutto. Se siamo in multi-processo e si blocca un processo, gli altri continuano ad andare e faccio ripartire normalmente quello che si è piantato.*

### Immagine che contiene testo, schermata, diagramma Descrizione generata automaticamente**Contesto Thread:**

In un processo posso avere più thread. Ogni processo ha il proprio codice, variabile globali e file che sono in comune per i suoi thread.

I thread possono girare in un multicore su cpu diverse e possono eseguire pezzi dello stesso programma.

## **Librerie Thread:**

La libreria dei thread mette a disposizione delle **funzioni per gestire i thread**.   
*Una thread library può essere disponibile a livello user o a livello kernel.*

* Thread sono **implementati** da una libreria di thread
* La libreria di thread salva i **contesti** dei thread quando questi non sono in esecuzione
  + **Salvati** in una struttura dati → per os161: thread structure

#### **Os161 thread structure**

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamenteContiene:

* un puntatore ad una struttura dati di sincronizzazione
* lo stack
  + kernel level stack
  + Void \* in quanto è un semplice indirizzo a memoria
* il contesto
  + Puntatore ad una struct in cui saranno salvati i registri
* il puntatore all cpu su cui gira il thread
* il puntatore al processo a cui appartiene il thread (nel caso in cui il thread appartenga al kernel, il processo sarà kernel)

Nota: negli user ci sono anche i thread ma potrei non aver le librerie per gestirle, noi ci focalizziamo su kernel.

**Immagine che contiene testo, schermata, diagramma, linea

Descrizione generata automaticamenteTHREAD LIBRARY E 2 THREADS**

Nel caso in cui si siano 2 thread e 1 CPU:

* Thread in esecuzione ha il suo stack e sta usando la CPU.
* Thead non attivo, è salvato, stack mantenuto in memoria e i suoi registri sono stati salvati.
  + Codice e dati kernel, con contesto salvato da qualche parte. (a dx)
* Immagine che contiene testo, diagramma, schermata, linea

  Descrizione generata automaticamenteDx codice thread
* Sx: struct thread
  + Ogni struct thread ha il puntatore in memoria al suo stack e al suo switch frame (che in os161 è nello stack).
  + Switchframe: contiene i registri che vanno salvati in cambio di contesto.

**Thrad 1 è in esecuzione** e infatti i suoi registri stanno in cpu.

**Thread 2** è in pausa e i suoi registri stanno nello switchframe.

### **Funzioni**

messe a disposizione dalla libreria sui thread:

* Immagine che contiene testo, ricevuta, Carattere, bianco

  Descrizione generata automaticamenteFunzioni per partenza/fine, usate quando servono

**Panic**→ gestire errori

**Shutdown**: chiude la gestione di tutti i thread

* Funzioni per gestione thread, chiamate al bootstrap

**Immagine che contiene testo, ricevuta, Carattere, bianco

Descrizione generata automaticamentefork** → crea thread

**exit**→ termina thread

**yield**→ termina esecuzione e fai posto a qualcun altro nella cpu, quindi cambi contesto

* Funzioni interne

**Immagine che contiene testo, ricevuta, Carattere, bianco

Descrizione generata automaticamentecreate**:

**destroy:** distrugge completamente (sarà chiamata da exit)

**make\_runnable**: schedula, lo mette in coda

**switch**: sarà chiamata da uield

#### **Thread fork**:

permette di creare un thread.

* Thread che chiama la thread fork: esegue una funzione che crea un nuovo thread e nelfrattempo lui continua a fare ciò che vuole fare.
* Per dire alla thread fork qual è **la funzione da eseguire**, si passa come parametro una funzione da eseguire passando un puntatore ad una funzione → **\*entrypoint**

**L**a funzione che va passata alla thread\_fork, deve essere consistente in nr di parametri e tipo → SONO FISSI.

* **Parametri**:
  + **Nome** thread
  + Puntatore al **processo** a cui il thread appartiene
  + **Funzione**
  + **void\*** → primo parametro della funzione : solitamente puntatore ad una struct, può essere anche un puntatore ad un inizio di un vettore
  + **unsigned long** → secondo parametro funzione
* ***Nota****:*
  + *In alcuni casi il thread appartiene al processo che li stanno creando ed è inutile passare il secondo parametro*
  + Immagine che contiene testo, schermata, Carattere, documento

    Descrizione generata automaticamente*In altri casi, il kernel crea una struttura dati processo e un kernel per farlo partire. In questo caso è molto inportante il seconod parametro*
* Esempio di chiamata:

Chiama o loudthread o quietthread 🡨

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

**Create**:Creato nuovo thread

**Switchframe**: Predispongo il contesto: puntatore a thread, funzione che devo eseguire e parametri.   
**makeRunnable**: metti in in coda alla lista della cpu per andare in esecuzione → READY.

**nota**: la fork mette il thread in coda ma non è detto che parti subito, partirà quando sarà chiamata la thread switch e thread sarà in testa.

#### **Thread switch:**

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente Funzione che **cambia il thread in esecuzione** sulla cpu:

* Riceve il nuovo stato del thread corrente (thread che deve abbandonare la cpu e lasciar posto ad un altro)
  + Nuovo stato== running → non va bene
  + Nuovo stato == ready → chiamo make\_runnabble che lo manda in coda
* Estrae dalla testa per prendere il nuovo thread
  + Chiamo switchframe\_switch che effettua il cambio di contesto, passando:
    - contesto del thread corrente → da salvare
    - contesto del thread nuovo → da ripristinare e quindi da usare (come puntatore by pointer)

*Nota: Thread yield chiama thread switch dicendogli che sono ready; thread switch ferma il thread in esecuzione e ne fa entrare un altro.*

* *Per far ciò la thread switch deve avere*
  + *Puntatore a struct thread corrente: curthread*
  + *Puntatore a strutct thread next*

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamenteKERNEL THREAD TEST:

**loud**: ci impiegano di più, fanno molti switch, hanno I/O

**quiet**: sono più veloci, fanno un input ad inizio e e un output alla fine

**Come avviene lo switch:** avviene il salvataggio dei registri del thread che viene sostituito e il ripristino dei registri del thread che viene ripristinato → decina di registri da 4 byte → **40 byte da salvare**.

* Immagine che contiene testo, schermata, Carattere, linea

  Descrizione generata automaticamenteSalvataggio dati vecchio thread:
  + Prendi stack
  + Sposta stack pointer di 40 byte
  + Puoi usare questi 40 byte
    - Salva i registri → salva il contesto
  + Sw sp, 0(a0) → sto mettondo context = sp
* Immagine che contiene testo, schermata, Carattere, documento

  Descrizione generata automaticamenteRipristino dati nuovo thread

Prendi il secondo parametro e modifica lo stackpointer → nuovo valore dello stackpointer per il nuovo context. Riprendi tutto e e inserisci nei registri.

Immagine che contiene testo, ricevuta, Carattere, documento

Descrizione generata automaticamente**nota**: alcuni registri sono salvati quando avviene una chiamata alla funzione, altri no

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

## Immagine che contiene testo, schermata, diagramma, linea Descrizione generata automaticamenteAPPLICAZIONE E KERNEL

NOTA: kernel ha propria libreria;   
processo user ha codice, dati e stack con registri (in os161 no heap tranne se qualcuno vuole implementare a mano la malloc).

Quando viene **creato un processo kernel**, ci sarà un processo user con struct proc:

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

→ processo sa quanti thread ha → **num threads**

→ha il gestore della memoria virtuale→ **address space**

***Nota****: non ha un puntatore ai suoi thread di default, ma si può implementare facendo un vettore (sovradimensionato o riallocabile) o una lista linkata*

**Immagine che contiene testo, schermata, diagramma, linea

Descrizione generata automaticamenteUn processo ha quindi**:

* 1 struttura dati proc
* 1 struttura dati thread per ogni thread

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente

**Nota**: quando un thread viene creato ha

* il proprio **stack di kernel** → dove vengono salvati i context quando si fa context switch)
* lo stack user dove ci sono variabili locali

*Each OS161 thread has two stacks, one that is used while the thread is executing unprivileged application code, and another that is used while the thread is executing privileged kernel code.*

## PROCESSO

File che contiene: codice, Variabili locali, strutture dati che metto in stack → eseguo processo e poi butto tutto.

**Processo user in esecuzione**, viene creato da un kernel thread che ha un suo stack, legge eseguibile, crea struttura dati allocando spazio di indirizzamento, carico eseguibile, thread in maniera privilegiata.

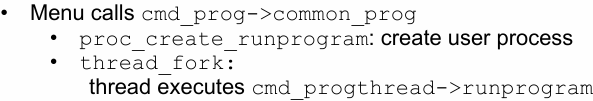
**Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamenteAddres space è una struct** che contiene:

* ptatori in memoria a 3 intervalli contigui:
  + 1 → fisica+logica
  + 2 → fisica+ logica
  + Stack

#### **CREARE PROCESSO** USER IN OS161:

* Immagine che contiene testo, Carattere, bianco

  Descrizione generata automaticamenteDal menu
* Menu effettua una catena di chiamate che arriva ad una **thread\_fork** → thread di kernel che viene lanciato con obbiettivo **cmd\_progthread** che ha un wrapper a runprogram.
  + Parte programma in esecuzione
  + Run program
    - Genera Spazio indirizzamento
    - Legge file formato elf
    - Kernel thread diventa uno user thread → nuovo processo

**Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente**Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente**runprogram**

**Trap frame**: struttura dati di salvataggio e ripristino in corrispondenza di una trap o itnerrupt.

Per far **partire un programma user**, lo si mette in una situazione in cui pensa di dover fare una trapFromInterrupt → per farlo partire, **sfrutto il protocollo di ritorno da interrupt**.

Quindi per far partire un programma: **enter\_new\_process**

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

*→ tf\_sp: stack pointer del processo user*

Gestendo come return for interrupt, lo faccio rornare in user mode.

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

*Nient’altro che return from exception*

Immagine che contiene testo, schermata, diagramma, numero

Descrizione generata automaticamenteProgramma parte in modalità user, ma programma in modalità user non può accedere a tutto, ad esempio dipositivi IO. **Dispositivi in IO vanno gestiti in modalità privilegiata**.

Quando c’è un programma user in esecuzione, il kernel fornisce una serie di servizi e ci sono le cosidette **system call** → interfaccia che il sistema operativo fornisce al programma utente per chiedere dei servizi.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamenteNota: se il nome del file è acquisito da tastiera:

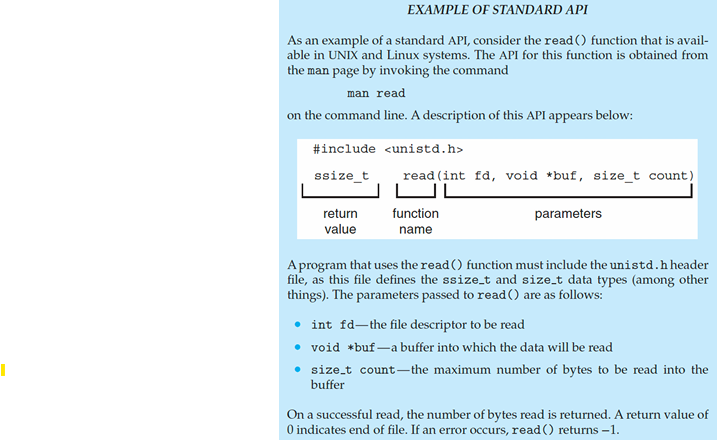
Slide 46 → esempio

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamenteSyscall:

* Riceve 55 (vedendo il file syscall.h, vedi che è una SysWrite)
* In syscall.c ho i capture degli

### **SystemCall**

Chiamate di sistema che reagiscono alle chiamate di un programma quando ha bisogno di un servizio dal sistema operativo.

Un programma user, per chiamare una funzione di sistema, deve chiamare delle funzioni privilegiate  
(non possono essere parte dell’eseguibile, programma user per usarle bisognerebbe fare una call a quel pezzo di kernel).   
→ Bisogna fare interrupt via sw (*es: quello che si attiva in caso di divisioni per 0 o provando ad usare un puntatore con valore 0 → TRAP: interrupt sw → stai chiamando qualcosa che non è un pezzo del tuo eseguibile*). Se l’applicazione user chiama open in user mode → in **kernel mode**: si consulta tabella in cui si comprende se si tratta di interrupt, trap o eccezione e poi si accede ad altra tabella che mi dice quale **funzione eseguire**:

Immagine che contiene testo, diagramma, schermata, Diagramma

Descrizione generata automaticamente

Dunque, possiamo dire che una call alla open non è altro che una **chiamata ad una trap**.

Come i metodi **passano parametri al sistemo operativo**?

* Semplice: passare i parametri ai **registri**
  + In alcuni casi ci possono essere più parametri di registri
* Parametri salvati in un **blocco o tabelle in memoria** e indirizzo del blocco passato come parametro del registro.

**TRAP FRAME**: x salvataggio e ritorno da una trap.

Anche il trap frame viene salvato nello stack, quando viene chiamata una sistem call:

* chiama sys\_call
* gestore delle trap che corrisponde alla system call
* arriva un trap
* syscall vede un puntatore al trapfram (vede i registri salvati).   
  Questi registri salvati vedono le cose che ci interessano