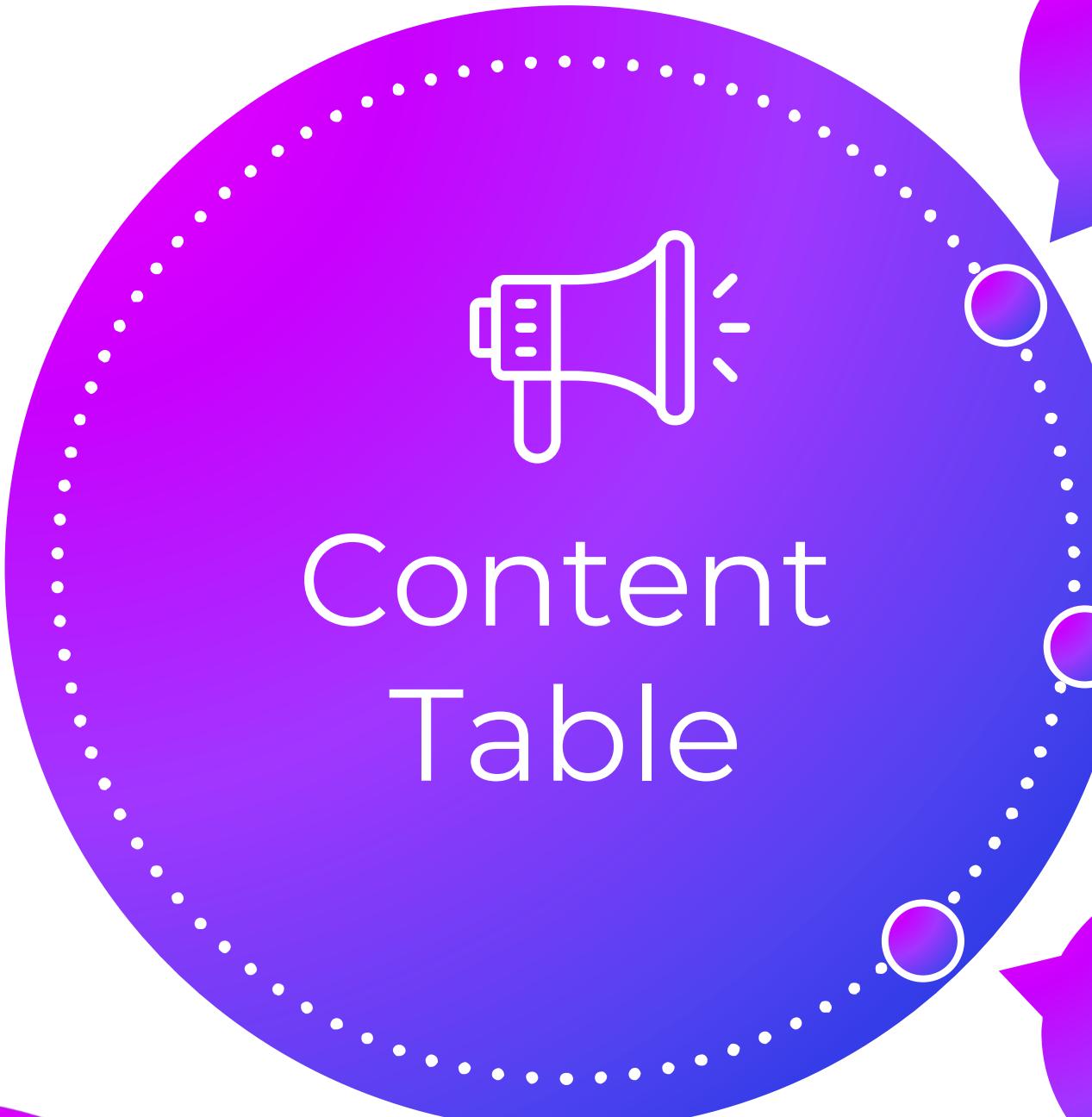


Progetto 1.1. Analisi Comparativa tra OS161 e altri Sistemi Operativi Open-Source

NachOs
vs
OS161

- 
- Team members**
-  Vittorio Sanfilippo s317408
 -  Marianna Francesca Amalfi s317407
 -  Salvatore Cavallaro s317842



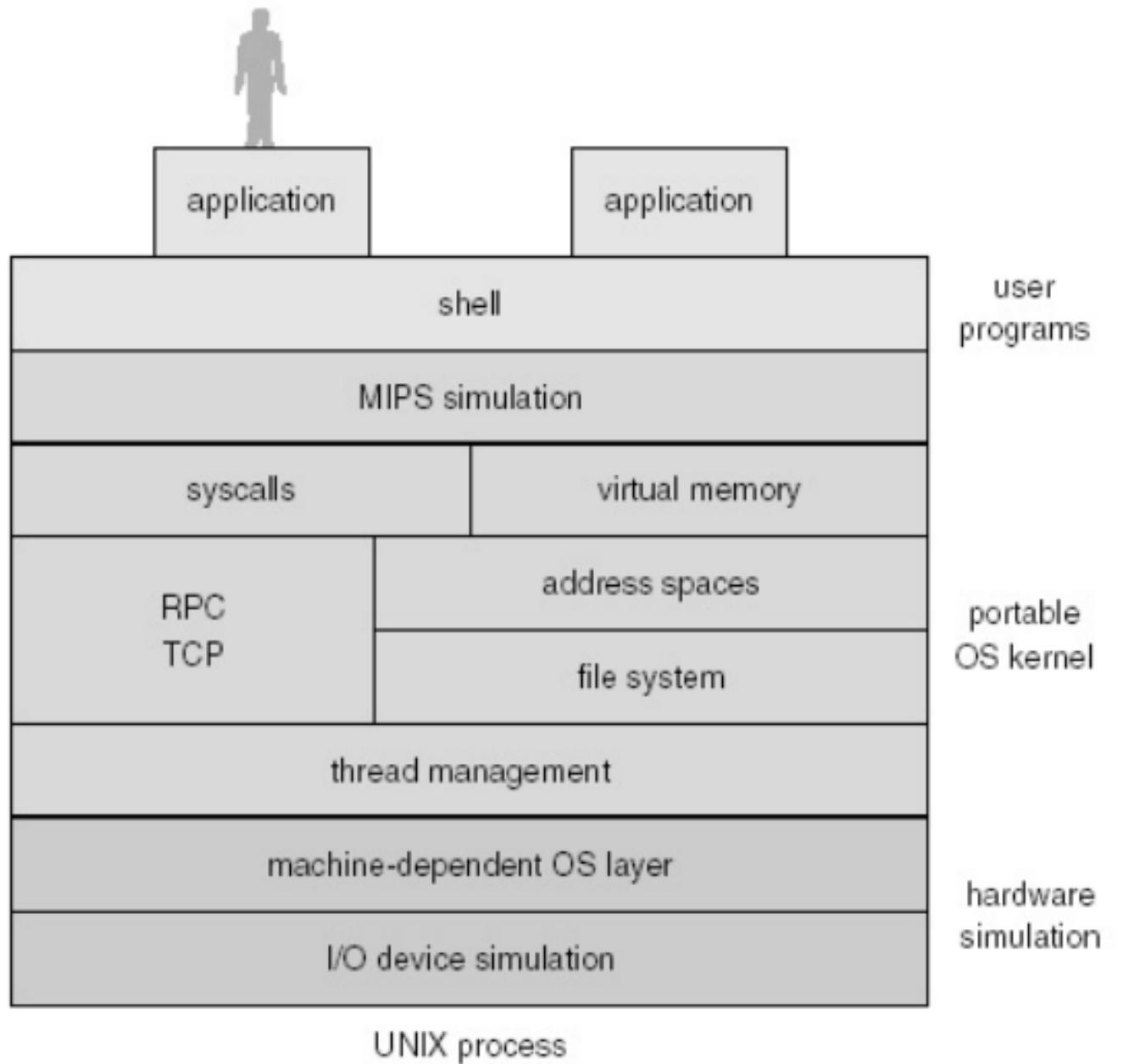
Content Table

- 01 **NachOs**
- 02 **Confronto NachOs
e Os161**
- 03 **Implementazione**

NachOS 3.4

Not Another Completely Heuristic Operating System, o NachOs, è un sistema operativo sviluppato presso l'Università della California, Berkeley, progettato da Thomas Anderson ed utilizzato a livello didattico per introdurre gli studenti ai concetti di progettazione e implementazione di sistemi operativi, richiedendo loro di implementare parti significative di funzionalità all'interno del sistema

Struttura



Il kernel viene eseguito al primo avvio di NachOs o quando un programma utente esegue un'istruzione che causa una hardware trap (ad esempio, un'istruzione illegale, un page fault, una chiamata di sistema, ecc.).

Il kernel è di tipo monolitico e viene eseguito sopra la simulazione hardware fornendo molte delle caratteristiche standard di un moderno kernel di sistema operativo, tra cui i thread, un sistema di file system e il supporto della memoria virtuale.

Le applicazioni a livello utente, come la shell, vengono eseguite al di sopra del kernel tramite una tradizionale system call interface

Originariamente scritto in C++ per MIPS, NachOs viene eseguito come processo utente su un sistema operativo host. Un simulatore MIPS esegue il codice dei programmi utente che girano sopra il sistema operativo NachOs.

Funzionalità Principali

Come OS/161, NachOs fornisce codice di basso livello per trappole e interrupt, driver di dispositivi, thread all'interno del kernel, uno scheduler di base e un sistema di memoria virtuale estremamente minimale. Include anche un semplice file system

Gestione Memoria

NachOs divide la memoria in due sezione :

- **User Space** : gestito dalla memoria virtuale , paginato
- **Kernel Space**: gestito dalla memoria fisica.

Riguardo lo **user Space**, la paginazione viene gestita o dalla TLB virtualizzata (Da attivare tramite flag USE_TLB) oppure tramite page table.

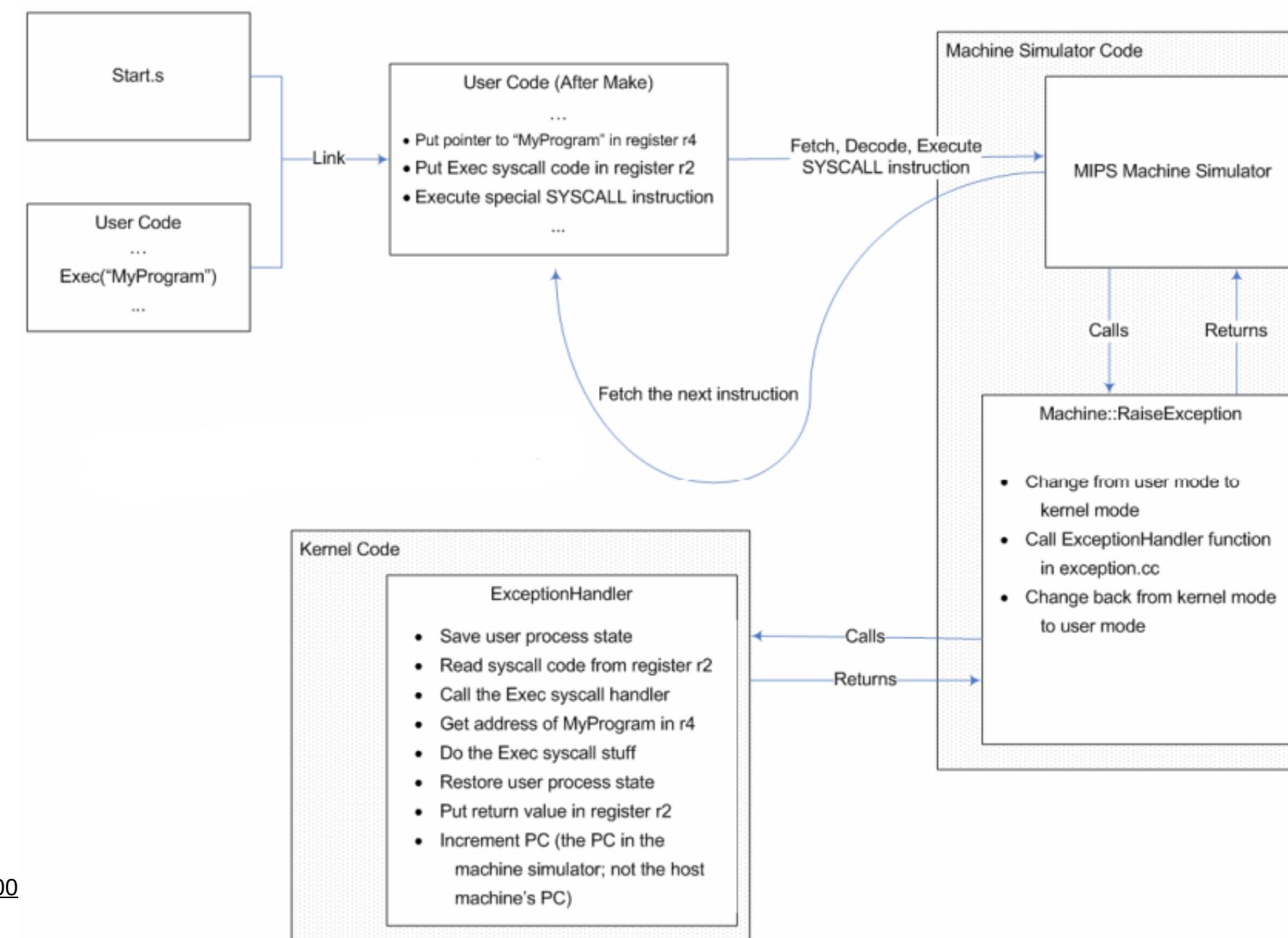
La Main Memory dedicata al processo utente (Machine-> mainMemory) è semplicemente un array di byte organizzata in pagine da 128 byte, la stessa dimensione dei settori del disco.

System calls

NachOs definisce solamente 10 tipi di system call differenti, ma ne viene implementata solo una , la Halt ().

La Machine (simulatore NachOs/MIPS) legge le varie istruzioni, se tra di loro trova OPCODE: OP_SYSCALL, viene chiamata la routine RaiseException() che switcha da usermode a kernel mode, e chiama ExceptionHandler() a cui passa il tipo di eccezione.

L'ExceptionHandler() gestisce tutti i tipi di eccezioni, quindi controlla se l'eccezione è una system call e chiama l'handler corretto.



File system

Il file system di NachOs è organizzato in directories , ne esistono due versioni:

- **Stub** : un front-end al filesystem Unix, in modo che gli utenti possano accedere ai file all'interno di NachOs senza dover scrivere il proprio filesystem
- **Real**: consente di implementare un filesystem Unix a cui si può accedere solo attraverso un disco simulato

Entrambi i file system forniscono lo stesso servizio e la stessa interfaccia

Nella implementazione **Real** abbiamo due strutture dati principali:

- **root directory** : Contiene tutti i file del file system, quindi nella versione base non c'è una struttura gerarchica.
Una directory è una tabella formata dalla coppia
Nome File - Sector number
- **Bitmap** : Serve per allocare i settori del disco e per mantenere l' informazione su quali settori sono liberi.

Ogni file nel file System ha :

- Header del file, salvato nel settore del disco
- Numero di blocchi di dato
- Un entry nella directory del file system

Il SynchDisk è un interfaccia con il disco che permettere la lettura e scrittura dei file in maniera “sincrona” (La richiesta aspetta finché non viene completata) . Il Disco a sua volta emula il comportamento di un disco reale, ha 32 settori ognuno contenente 32 tracce

source: <https://users.cs.duke.edu/~narten/110/nachos/main/node22.html>
source: <https://users.cs.duke.edu/~narten/110/nachos/main/node3.html>

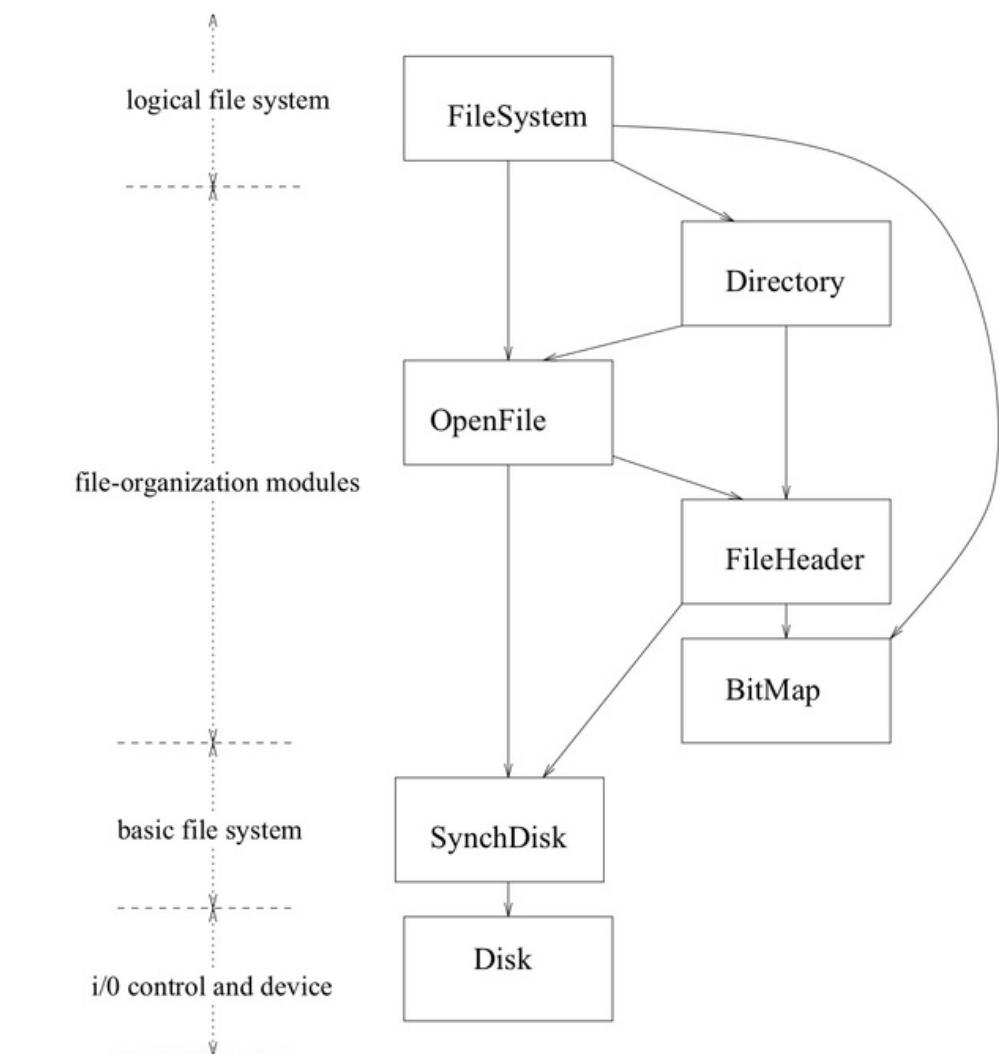


Figure 10.1: Structure of the Nachos File System

Thread

In NachOS viene definita una classe di Thread. Un thread ha uno stato associato che può essere ready, running, blocked o created. L'oggetto thread ha vari metodi come PutThreadToSleep, YieldCPU, ThreadFork, ThreadStackAllocate, ecc. Ogni thread viene eseguito in uno spazio di indirizzi virtuali.

A differenza di altri sistemi, NachOs non mantiene una tabella dei processi esplicita. Invece, le informazioni associate ai thread sono mantenute come dati (solitamente) privati di un'istanza dell'oggetto Thread. NachOs sparge le sue "thread table entries" in tutta la memoria, infatti per ottenere le informazioni di un thread specifico è necessario un puntatore all'istanza del thread.

Sincronizzazione

Le procedure di basso livello di NachOs disabilitano e riabilitano frequentemente gli interrupt per ottenere la mutua esclusione.

Come primitive di sincronizzazione troviamo solamente i semafori, già definiti e implementati nel sistema.

Lock e Condition Variable sono definite ma la loro implementazione è lasciata agli studenti.

Processi

NachOs esegue i processi nel loro address space privato. Permette di eseguire qualsiasi MIPS binario, effettuando una traduzione dal formato Coff a quello Noff. Un processo viene creato insieme ad un address space e viene anche caricato il contenuto dell'eseguibile nella main memory e invocata la macchina NachOs. Ogni Processo ha bisogno di almeno un thread, con cui condivide lo stesso Address Space.

La macchina NachOs/MIPS è implementata dall'oggetto Machine, una cui istanza viene creata al primo avvio di NachOs. L'oggetto Machine esporta una serie di operazioni e variabili pubbliche a cui il kernel NachOs accede direttamente, fornisce registri, memoria fisica, supporto alla memoria virtuale e operazioni per eseguire la macchina o esaminare il suo stato corrente.

```
42
43 void
44 StartProcess(char *filename)
45 {
46     OpenFile *executable = fileSystem->Open(filename);
47     AddrSpace *space;
48
49     if (executable == NULL) {
50         printf("Unable to open file %s\n", filename);
51         return;
52     }
53     space = new AddrSpace(executable);
54     currentThread->space = space;
55
56     delete executable;           // close file
57
58     space->InitRegisters();    // set the initial register values
59     space->RestoreState();     // load page table register
60
61     machine->Run();           // jump to the user program
62     ASSERT(FALSE);            // machine->Run never returns;
63                                         // the address space exits
64                                         // by doing the syscall "exit"
65 }
66
```

NachOS vs Os161

NachOs

- **Kernel:** Monolitico
- **Architettura:** NachOS utilizza un'architettura "mixed mode" in cui il kernel e il simulatore della macchina sono compilati in un singolo eseguibile. Originariamente scritto in C++ per MIPS. Un simulatore MIPS esegue il codice dei programmi utente che girano sopra il sistema operativo NachOS
- **Complessità:** NachOS utilizza un approccio di esecuzione mista per il debugging.

Os161

- **Kernel:** Macrokernel
- **Architettura:** OS/161 utilizza un approccio più tradizionale con un ambiente di esecuzione simulato (System/161) separato dal kernel (OS/161). È scritto in C e include il supporto per una sola architettura, ma è strutturato come un sistema portatile, con codice dipendente dal processore e dalla piattaforma separato per la manutenzione.
- **Complessità:** OS/161 è più strutturato e complesso. OS/161 include un supporto di debugging remoto con gdb.

NachOS vs Os161

NachOs

- **Gestione Memoria:** User Space paginato ma è possibile anche attivare la TLB virtualizzata. NachOs divide la memoria in due sezione :
 - User Space : gestito dalla memoria virtuale , paginato
 - Kernel Space: gestito dalla memoria fisica
- **Sincronizzazione:** Interrupt e Semafori presenti, Lock e Condition Variable sono supportati ma necessitano di essere implementati

Os161

- **Gestione Memoria:** Gestisce la memoria tramite allocazione contigua ma è possibile utilizzare anche la paginazione(tramite page table o implementazione TLB). consiste in due elementi :
 - Kmalloc: permette l'allocazione della memoria del kernel
 - Dumbvm: gestisce la memoria dello spazio utente
- **Sincronizzazione:** Interrupt, Semafori, SpinLock e wait channel presenti , Lock e Condition Variable sono supportati ma necessitano di essere implementati

NachOS vs Os161

NachOs

- **File System:** Due versioni: Stub e Real
Nella implementazione Real abbiamo due strutture dati principali:
 - root directory: contiene tutti i file del file system
 - Bitmap : serve per allocare i settori del disco e mantenere l' informazione su quali sono liberi.
- **System Call:** Gestite tramite l'ExceptionHandler() generico che gestisce tutti i tipi di eccezioni, quindi controlla se l'eccezione è una system call e chiama l'handler corretto.

Os161

- **File System:**
 - SFS(Simple File System): Un esempio di un file system basico.
 - VFS (Virtual File System):Un'interfaccia standard che permette ai processi di interagire con il file system indipendentemente dal file system sottostante.
- **System Call:** Gestite nell'apposito system call handler. Si estrae il valore della systemcall dal registro v0, questo verrà utilizzato all'interno di uno switch case per andare nell'apposito “systemcall handler”

NachOS vs Os161

NachOs

- **Threads:** In NachOs viene definita una classe di thread, non viene creata una thread table entries. NachOs sparge le sue "thread table entries" in tutta la memoria, infatti per ottenere le informazioni di un thread specifico è necessario un puntatore all'istanza del thread.
- **Processi:** NachOs esegue i processi nel loro address space privato. Permette di eseguire qualsiasi MIPS binario. Ad ogni processo viene assegnato un thread con lo stesso address space. Quando un processo viene creato, viene creato un address space, caricato il contenuto dell'eseguibile nella main memory e invocata la macchina NachOs.

Os161

- **Threads:** Vengono implementati da una libreria e aggiunti e gestiti da una threadList. La libreria salva il contesto in una struttura chiamata Thread Control Block, salvato nella struttura del thread. In essa è possibile notare:
 - Lo stato del thread
 - il puntatore allo stack
 - il puntatore allo Switchframe
 - il puntatore al processo a cui è associato
- **Processi:** Per ogni programma viene creato un processo e la struttura PCB. Ogni processo ha bisogno di almeno un thread. Nella struttura del processo troviamo:
 - il numero di thread che sono associati al processo
 - un puntatore alla virtual address space, cioè dove il processo è mappato nella memoria virtuale
 - uno spinlock sulla struttura stessa.

Implementazione

- Modifica della struttura dei thread.
- Aggiunta di un algoritmo di scheduling
- Implementazione dei Lock e delle Condition Variable

Modifica della struttura dei thread

Le Modifiche riguardanti la struttura dei thread riguardano principalmente l' introduzione di :

- un campo **tid**, che permette di identificare un thread .
- un campo **uid**, che consente di associare un thread a un processo utente
- un campo **priority**, che definisce la priorità per quello specifico thread.

Inoltre nel sistema sono stati introdotti due array che favoriscono la gestione dei thread, nello specifico è stato introdotto il **tid_flag** che mantiene l' informazione dei thead creati tramite un booleano nei vari campi dell' array, e **tid_pointer** che salva i puntatori alle strutture dei thread .

```
76 class Thread {
77     private:
78         // NOTE: DO NOT CHANGE the order of these first two members.
79         // THEY MUST be in this position for SWITCH to work.
80         int* stackTop;      // the current stack pointer
81         void *machineState[MachineStateSize]; // all registers except for stackTop
82         int tid;
83         int uid;
84
85         int priority;
86
87     public:
88         int getThreadId() { return (tid); }
89         int getUserId() { return (uid); }
90         int getPriority() {return(priority);}
91
92     ThreadStatus getThreadStatus(){ return (status);}
93
94     void setUserId(int userId) { uid = userId; }
95     void setPriority (int p ) {priority = p; }
96
97     Thread (char* debugName, int p ); // initialize a Thread with priority
98     Thread(char* debugName); // initialize a Thread
99     ~Thread(); // deallocate a Thread
100    // NOTE -- thread being deleted
101    // must not be running when delete
102    // is called
```

```
34 Thread::Thread(char* threadName)
35     : Thread(threadName,0)
36 {}
37
38 Thread::Thread(char* threadName, int p )
39     : priority (p)
40 {
41
42     bool success_allocate = FALSE;
43     for (int i = 0; i < MAX_THREAD_NUM; i++) { // sequential search
44         if (!tid_flag[i]) { // if found an empty space
45             this->tid = i;
46             tid_flag[i] = TRUE;
47             tid_pointer[i]=this;
48             success_allocate = TRUE;
49             break;
50         }
51     }
52
53
54     if (!success_allocate) {
55         printf("Reach maximum threads number %d, unable to allocate!!", MAX_THREAD_NUM);
56     }
57     ASSERT(success_allocate); // abort if unable to allocate new thread
58     name = threadName;
59     stackTop = NULL;
60     stack = NULL;
61     status = JUST_CREATED;
62 #ifdef USER_PROGRAM
63     space = NULL;
64 #endif
65 }
66
67 //-----
68 // Thread::Thread
69 // De-allocate a thread.
70 //
71 // NOTE: the current thread *cannot* delete itself directly,
72 // since it is still running on the stack that we need to delete.
73 //
74 // NOTE: if this is the main thread, we can't delete the stack
75 // because we didn't allocate it -- we got it automatically
76 // as part of starting up Nachos.
77 //
78
79 Thread::Thread()
80 {
81     ASSERT(this != currentThread);
82     tid_flag[this->tid] = FALSE;
83     tid_pointer[this->tid]=NULL;
84     DEBUG('t', "Deleting thread \'%s\'\n", name);
85
86     if (stack != NULL)
87         DeallocBoundedArray((char *) stack, StackSize * sizeof(int));
88
89 }
```

Aggiunta di un algoritmo di scheduling

Grazie all' aggiunta del campo **priority** all'interno della struttura del Thread possiamo sostituire alla semplice coda FIFO di NachOs sempre una coda FIFO ma con priorità. È possibile assegnare facilmente la priorità utilizzando la modifica fatta ai costruttori vista in precedenza

```
void
TestPriority()
{
    DEBUG('t', "Entering TestPriority");

    Thread *t1 = new Thread("with p", 87);

    Thread *t2 = new Thread("set p");
    t2->setPriority(100);

    Thread *t3 = new Thread("no p");

    t1->Fork(CustomThreadFunc, (void*)0);
    t2->Fork(CustomThreadFunc, (void*)0);
    t3->Fork(CustomThreadFunc, (void*)0);

    CustomThreadFunc(0); // Yield the current thread

    printf("--- Calling TS command ---\n");
    TS();
    printf("--- End of TS command ---\n");
}
```

```
lubuntu@lubuntu-VirtualBox: ~/Scriv.../Nachos3.4/nachos-3.4/code/threads - + ×
File Modifica Schede Aiuto
Q Q
*** current thread (uid=0, tid=0, priority= 0, name=main) => Yield
Ready list contents:
no p, with p, set p,
*** current thread (uid=0, tid=3, priority= 0, name=no p) => Yield
Ready list contents:
main, with p, set p,
--- Calling TS command ---
UID      TID      NAME      PRI      STATUS
0        0        main      0        RUNNING
0        1        with p   87      READY
0        2        set p    100     READY
0        3        no p     0        READY
--- End of TS command ---
*** current thread (uid=0, tid=1, priority= 87, name=with p) => Yield
Ready list contents:
set p,
*** current thread (uid=0, tid=2, priority= 100, name=set p) => Yield
Ready list contents:
with p,
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
Ticks: total 110, idle 0, system 110, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
...
```

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());

    thread->setStatus(READY);
    //readyList->Append((void *)thread);
    // i Thread vengono aggiunti alla ready list in base alla loro priorità
    readyList->SortedInsert((void*)thread, thread->getPriority());
}
```

Aggiunta di un algoritmo di scheduling

In aggiunta alla modifica della coda FIFO di NachOs abbiamo implementato l' algoritmo **Round Robin**.

Essendo un algoritmo preemptive per prima cosa abbiamo deciso di implementare un timer che viene attivato da uno specifico flag inserito tramite riga di comando. Alla scadenza del timer verrà scatenato un interrupt che verrà gestito dalla funzione **RRHandler**.

```
128 // Round robin
129 bool roundRobin = FALSE;
130 else if (!strcmp(*argv, "-rr")) { //activate RR timer
131     ASSERT(argc > 1);
132     roundRobin = TRUE;
133     argCount = 2;
134 }
135
136
104
165 //ROUND ROBIN
166 if (roundRobin) // fa partire il timer RR
167 timer = new Timer(RRHandler, 0, FALSE);
168
```

L'handler si occupa riordinare la Ready List del sistema. Quando viene chiamata controlla se il numero di ticks del processo è maggiore o uguale a 100, in tal caso viene effettuato un Contex Switch per far entrare il thread successivo.

```
156 static void RRHandler(int dummy)
157 {
158     // scheduler->lastSwitchTick salva l' informazione del numero tick al precedente context Switching
159     //TimerTicks è una variabile globale del sistema che vale 100 ticks
160
161     int timeDuration = stats->totalTicks - scheduler->lastSwitchTick;
162     printf("\nTimer interrupt with duration: %d", timeDuration);
163     //se il timeDuration è maggiore del TimerTicks vuol dire che il tempo disponibile per il processo è terminato,
164     // quindi bisogna fare un context switch in modo da far entrare il prossimo Thread
165     if (timeDuration >= TimerTicks) {
166         if (interrupt->getStatus() != IdleMode) { // se il sistema è in IdleMode la ready List è vuota
167             printf(" (Determine to Context switch)\n");
168             interrupt->YieldOnReturn();
169             scheduler->lastSwitchTick = stats->totalTicks; // aggiorna lastSwitchTick
170         } else {
171             printf(" (readyList is Empty)\n");
172         }
173     } else {
174         printf("\n");
175     }
176 }
```

Aggiunta di un algoritmo di scheduling

Funzione di Test

Per la funzione di test abbiamo creato 3 Thread con differenti burst time(running time).

Viene fatta una fork che porta il processo figlio ad eseguire la funzione ThreadWithTick che semplicemente serve a far andare avanti il sistema di un tick.

Di default se il thread chiamante è un thread di sistema , la funzione oneTick () incrementerà i ticks del sistema di 10. Quindi prima di avere un context Switch il thread avrà a disposizione 9 cicli + 1 per lo switch

```
283 void
284 ThreadWithTicks(int runningTime)
285 {
286     int num;
287
288     for (num = 0; num < runningTime * SystemTick; num++) {
289         printf("### thread with running time %d looped %d times (stats->totalTicks: %d)\n",
290               runningTime, num+1, stats->totalTicks);
291         interrupt->OneTick(); // make system time moving forward (advance simulated time)
292     }
293     currentThread->Finish();
294 }
295 -----
296 // RoundRobinTest
297 // Fork some Thread with different amount of bursts
298 // and observe if the lower one will take over the CPU
299 //-----
300
301 void
302 RoundRobinTest()
303 {
304     DEBUG('t', "Entering RoundRobinTest");
305
306     printf("\nSystem initial ticks:\tsystem=%d, user=%d, total=%d\n", stats->systemTicks, stats->userTicks, stats->totalTicks);
307
308     Thread *t1 = new Thread("7");
309     Thread *t2 = new Thread("2");
310     Thread *t3 = new Thread("5");
311
312     printf("\nAfter new Thread ticks:\tsystem=%d, user=%d, total=%d\n", stats->systemTicks, stats->userTicks, stats->totalTicks);
313
314     t1->Fork(ThreadWithTicks, (void*)7);
315     t2->Fork(ThreadWithTicks, (void*)2);
316     t3->Fork(ThreadWithTicks, (void*)5);
317
318     printf("\nAfter 3 fork() ticks:\tsystem=%d, user=%d, total=%d\n\n", stats->systemTicks, stats->userTicks, stats->totalTicks);
319
320     // update the lastSwitchTick
321     scheduler->lastSwitchTick = stats->totalTicks;
322     currentThread->Yield(); // Yield the main thread
323 }
```


Lock e Condition Variable

Lock

Per l' implementazione dei Lock ci siamo rifatti al Lab03 di Os161.

Per la struttura abbiamo sfruttato i semafori già presenti nel sistema , andandone a inserire uno binario, in modo da non dover gestire il wait e signal per l'Acquire del Lock.

Di lato l' implementazione delle funzioni del **Lock**.

Da attenzionare particolarmente, specialmente nella fuzione di *Release()*, dove grazie alla funzione *isHeldByCurrentThread()* andiamo a controllare se il thread che sta rilasciando il lock sia effettivamente lo stesso che lo ha acquisito.

Così facendo andiamo a implementare la carettiristica principale del lock, cioè il concetto di **possesso**.

```
65
66 class Lock {
67 public:
68     Lock(char* debugName);      // initialize lock to be FREE
69     ~Lock();                  // deallocate lock
70     char* getName() { return name; } // debugging assist
71
72     void Acquire(); // these are the only operations on a lock
73     void Release(); // they are both *atomic*
74
75     bool isHeldByCurrentThread(); // true if the current thread
76     // holds this lock. Useful for
77     // checking in Release, and in
78     // Condition variable ops below.
79
80 private:
81     char* name;           // for debugging
82     // plus some other stuff you'll need to define
83
84     //Custom
85     Thread * lock_owner; //Pointer to the thread that acquired the lock
86     Semaphore *semaphore;
87 };
88
```

```
103 Lock::Lock(char* debugName) {
104     name = debugName;
105     semaphore= new Semaphore("Lock",1);
106     lock_owner= NULL ;
107 }
108 Lock::~Lock() {
109     delete semaphore;
110 }
111 void Lock::Acquire() {
112     //controllo che il lock non sia già acquisito
113     ASSERT(! this -> isHeldByCurrentThread());
114
115     //mascheriamo gli interrupt per evitare che il thread che sta acquisendo il lock venga tolto dalla
116     IntStatus oldLevel = interrupt->SetLevel(IntOff);
117
118     DEBUG ('s', "Lock %s" Acquired by Thread \"%s\"\n", name , currentThread->getName());
119
120     //utilizziamo il semaforo binario per gestire l'acquisizione o l'eventuale wait
121     semaphore->P();
122     lock_owner= currentThread;
123
124     //abilitiamo nuovamente gli interrupt
125     (void) interrupt -> SetLevel (oldLevel);
126 }
127 void Lock::Release() {
128     //controllo che il lock che sta facendo la release sia effettivamente l' owner del lock
129     ASSERT( this -> isHeldByCurrentThread());
130
131     //disabilito gli interrupt
132     IntStatus oldLevel = interrupt->SetLevel(IntOff);
133
134     DEBUG ('s', "Lock %s" Released by Thread \"%s\"\n", name , currentThread->getName());
135     lock_owner = NULL ;
136
137     semaphore->V();
138
139     //abilitiamo gli interrupt
140     (void) interrupt -> SetLevel (oldLevel);
141 }
142
143 bool Lock::isHeldByCurrentThread(){
144     return lock_owner==currentThread;
145 }
```

Lock e Condition Variable

Condition Variable

Riguardo le Condition Variable nella struttura abbiamo aggiunto una semplice lista di thread che rappresenta la lista dove i thread aspetteranno.

Per l' implementazione effettiva delle "dummy functions" abbiamo utilizzato il Lock, che ne garantisce un accesso protetto in mutua esclusione.

Come funzioni principali troviamo :

- **Wait**: permette al thread di rilasciare il lock e andare in sleep , per poi riacquisirlo una volta svegliato.
- **Signal e Broadcast**: svolgono il compito di risvegliare il thread. La prima risveglia solo il primo thread in coda, mentre la seconda li sveglia tutti.

Da notare che per ogni funzione vengono disabilitati gli interrupt, in modo che venga evitato il context Switch mentre si sta svolgendo tale operazione.

```
121 class Condition {
122 public:
123     Condition(char* debugName); // initialize condition to
124     // "no one waiting"
125     ~Condition(); // deallocate the condition
126     char* getName() { return (name); }
127
128     void Wait(Lock *conditionLock); // these are the 3 operations on
129     // condition variables; releasing the
130     // lock and going to sleep are
131     // *atomic* in Wait()
132     void Signal(Lock *conditionLock); // conditionLock must be held by
133     void Broadcast(Lock *conditionLock); // the currentThread for all of
134     // these operations
135
136 private:
137     char* name;
138     //custom
139     List* queue; //Wait list
140     // plus some other stuff you'll need to define
141 };
142
143 Condition::~Condition() { }
144 void Condition::Wait(Lock* conditionLock) {
145     //ASSERT(FALSE);
146     //disabilitiamo gli interrupt
147     IntStatus oldLevel = interrupt->SetLevel(IntOff);
148
149     //Verifichiamo che il Thread che sta andando in sleep sia l'owner del lock
150     ASSERT( conditionLock -> isHeldByCurrentThread());
151
152     //Rilasciamo il lock prima di andare in sleep
153     conditionLock->Release();
154
155     //inseriamo il thread nella coda di sleep
156     queue->Append(currentThread);
157     currentThread->Sleep();
158
159     //Dopo l'awake il Thread riacquista il lock
160     conditionLock->Acquire();
161
162     (void) interrupt -> SetLevel (oldLevel);
163 }
164 void Condition::Signal(Lock* conditionLock) {
165     //disabilito gli interrupt
166     IntStatus oldLevel = interrupt->SetLevel(IntOff);
167
168     ASSERT( conditionLock -> isHeldByCurrentThread());
169
170     //sveglio il primo Thread in coda, se ce n'è qualcuno
171
172     if(!queue->IsEmpty()){
173         Thread *nextThread = queue->Remove();
174
175         scheduler->ReadyToRun(nextThread);
176     }
177
178     (void) interrupt -> SetLevel (oldLevel);
179 }
180
181 void Condition::Broadcast(Lock* conditionLock) {
182     //disabilito gli interrupt
183     IntStatus oldLevel = interrupt->SetLevel(IntOff);
184
185     ASSERT( conditionLock -> isHeldByCurrentThread());
186
187     //sveglio tutti i thread in coda
188
189     while(!queue->IsEmpty()){
190         Signal(conditionLock);
191     }
192
193     (void) interrupt -> SetLevel (oldLevel);
194 }
```

Lock e Condition Variable

Programma di Test- Esempio Produttore/Consumatore

Abbiamo creato esternamente alle funzioni due CV, una per il consumatore e una per il produttore in modo che ognuno abbia la propria coda di wait. Inoltre è stato aggiunto un lock sulla risorsa per avere un accesso sicuro .

```
328 //-----  
329 //Functions to test Lock and Condition Variable implementation  
330 //-----  
331  
332 Condition * cond_Consumer = new Condition ("Consumer Condition");  
333 Condition * cond_Producer = new Condition ("Producer Condition");  
334  
335 Lock *resourceLock = new Lock ("Lock");  
336  
337 int resource = 0 ;  
338 int limit = 10 ;  
339
```

Produttore

Il thread produttore quando entra nella funzione cercherà di acquisire il Lock della risorsa, una volta acquisito controllerà se la risorsa ha raggiunto il limite massimo. Se il limite è raggiunto andrà in sleep liberando il lock, altrimenti incrementerà la risorsa di uno, farà il signal al thread consumatore e poi libererà il lock.

```
340 //-----  
341 //-----  
342 //The producer thread will try to access the resource,  
343 //if it has reached the limit it will go to sleep,  
344 // otherwise it will increment the resource and send a signal to the consumer thread  
345 //-----  
346  
347 void Producer (){  
348     for(int i = 0; i < 30 ; i++){  
349         resourceLock->Acquire();  
350  
351         while (resource>= limit){  
352             printf("Le risorse hanno raggiunto il limite, il Produttore %s va in sleep \n", currentThread->getName());  
353             cond_Producer->Wait(resourceLock);  
354         }  
355  
356         ++resource;  
357         printf("Il produttore %s ha aggiunto una risorsa, tot %d\n", currentThread->getName(), resource);  
358  
359         cond_Consumer->Signal(resourceLock);  
360         resourceLock->Release();  
361     }  
362 }
```

Lock e Condition Variable

Programma di Test- Esempio Produttore/Consumatore

Consumatore

Il thread Consumatore quando entra nella funzione cercherà di acquisire il Lock della risorsa, una volta acquisito controllerà se la risorsa ha lo zero. Se è stato raggiunto andrà in sleep liberando il lock, altrimenti decrementerà la risorsa di uno, farà il signal al thread produttore e poi libererà il lock.

```
364 //-----
365 //-----  
366 //Consumer  
367 //The consumer thread will try to access the resource,  
368 // if it is 0 it will go to sleep,  
369 //otherwise it will decrement the resource and send a signal to the producer Thread//  
370 //-----  
371  
372 void Consumer (){  
373  
374     for(int i = 0; i < 30 ; i++){  
375         resourceLock->Acquire();  
376  
377         while (resource<= 0){  
378             printf("Le risorse sono esaurite, il Consumatore %s va in sleep\n ", currentThread->getName());  
379             cond_Consumer->Wait(resourceLock);  
380         }  
381  
382         --resource;  
383         printf("Il consumatore %s ha rimosso una risorsa, tot %d\n", currentThread->getName(), resource);  
384  
385         cond_Producer->Signal(resourceLock);  
386         resourceLock->Release();  
387     }  
388 }
```

Funzione di Test

Nella funzione di test sono stati semplicemente creati due thread per “Categoria” e poi è stata fatta un fork sulla funzione di competenza

```
390  
391 void SyncTest (){  
392  
393     DEBUG('t', "Entering in Sync test ");  
394  
395     Thread *p1 = new Thread("Produttore 1 ");  
396     Thread *p2 = new Thread("Produttore 2 ");  
397  
398     Thread *c1 = new Thread("Consumatore 1 ");  
399     Thread *c2 = new Thread("Consumatore 2 ");  
400  
401     p1->Fork(Producer, nullptr);  
402     c1->Fork(Consumer, nullptr);  
403     c2->Fork(Consumer, nullptr);  
404     p2->Fork(Producer, nullptr);  
405  
406 }  
407
```

Lock e Condition Variable

Programma di Test- Esempio Produttore/Consumatore

Output

```
lubuntu@lubuntu-VirtualBox: ~/Scrivania/Nachos3.4/nachos-3.4/code/threads
File Modifica Schede Aiuto

lubuntu@lubuntu-VirtualBox:~/Scrivania/Nachos3.4/nachos-3.4/code/threads$ ./nachos -rs -q 4
Il produttore Produttore 1 ha aggiunto una risorsa, tot 1
Il produttore Produttore 1 ha aggiunto una risorsa, tot 2
Il produttore Produttore 1 ha aggiunto una risorsa, tot 3
Il produttore Produttore 1 ha aggiunto una risorsa, tot 4
Il produttore Produttore 1 ha aggiunto una risorsa, tot 5
Il produttore Produttore 1 ha aggiunto una risorsa, tot 6
Il consumatore Consumatore 1 ha rimosso una risorsa, tot 5
Il consumatore Consumatore 1 ha rimosso una risorsa, tot 4
Il consumatore Consumatore 1 ha rimosso una risorsa, tot 3
Il consumatore Consumatore 1 ha rimosso una risorsa, tot 2
Il consumatore Consumatore 1 ha rimosso una risorsa, tot 1
Il consumatore Consumatore 1 ha rimosso una risorsa, tot 0
Le risorse sono esaurite, il Consumatore Consumatore 2 va in sleep
Il produttore Produttore 1 ha aggiunto una risorsa, tot 1
Il produttore Produttore 1 ha aggiunto una risorsa, tot 2
Il produttore Produttore 1 ha aggiunto una risorsa, tot 3
Il produttore Produttore 1 ha aggiunto una risorsa, tot 4
Il produttore Produttore 1 ha aggiunto una risorsa, tot 5
Il produttore Produttore 1 ha aggiunto una risorsa, tot 6
Il produttore Produttore 1 ha aggiunto una risorsa, tot 7
Il produttore Produttore 1 ha aggiunto una risorsa, tot 8
Il produttore Produttore 1 ha aggiunto una risorsa, tot 9
Il produttore Produttore 1 ha aggiunto una risorsa, tot 10
Le risorse hanno raggiunto il limite, il Produttore Produttore 1 va in sleep
Le risorse hanno raggiunto il limite, il Produttore Produttore 2 va in sleep
Il consumatore Consumatore 1 ha rimosso una risorsa, tot 9
Il consumatore Consumatore 1 ha rimosso una risorsa, tot 8
Il consumatore Consumatore 1 ha rimosso una risorsa, tot 7
Il consumatore Consumatore 2 ha rimosso una risorsa, tot 6
Il consumatore Consumatore 2 ha rimosso una risorsa, tot 5
Il consumatore Consumatore 2 ha rimosso una risorsa, tot 4
Il consumatore Consumatore 2 ha rimosso una risorsa, tot 3
Il consumatore Consumatore 2 ha rimosso una risorsa, tot 2
Il consumatore Consumatore 2 ha rimosso una risorsa, tot 1
Il consumatore Consumatore 2 ha rimosso una risorsa, tot 0
Le risorse sono esaurite, il Consumatore Consumatore 2 va in sleep
Il produttore Produttore 1 ha aggiunto una risorsa, tot 1
Il produttore Produttore 1 ha aggiunto una risorsa, tot 2
Il produttore Produttore 1 ha aggiunto una risorsa, tot 3
Il produttore Produttore 2 ha aggiunto una risorsa, tot 4
Il consumatore Consumatore 2 ha rimosso una risorsa, tot 3
Il consumatore Consumatore 2 ha rimosso una risorsa, tot 2
Il consumatore Consumatore 2 ha rimosso una risorsa, tot 1
Il consumatore Consumatore 2 ha rimosso una risorsa, tot 0
Le risorse sono esaurite, il Consumatore Consumatore 2 va in sleep
Il produttore Produttore 1 ha aggiunto una risorsa, tot 1
Il consumatore Consumatore 1 ha rimosso una risorsa, tot 0
Il produttore Produttore 2 ha aggiunto una risorsa, tot 1
Il produttore Produttore 2 ha aggiunto una risorsa, tot 2
Il produttore Produttore 2 ha aggiunto una risorsa, tot 3
Il consumatore Consumatore 2 ha rimosso una risorsa, tot 2
Il consumatore Consumatore 2 ha rimosso una risorsa, tot 1
Il produttore Produttore 1 ha aggiunto una risorsa, tot 2
Il produttore Produttore 1 ha aggiunto una risorsa, tot 3
Il produttore Produttore 1 ha aggiunto una risorsa, tot 4
Il produttore Produttore 1 ha aggiunto una risorsa, tot 5
Il produttore Produttore 1 ha aggiunto una risorsa, tot 6
Il produttore Produttore 1 ha aggiunto una risorsa, tot 7
```



**Grazie per
l'attenzione**