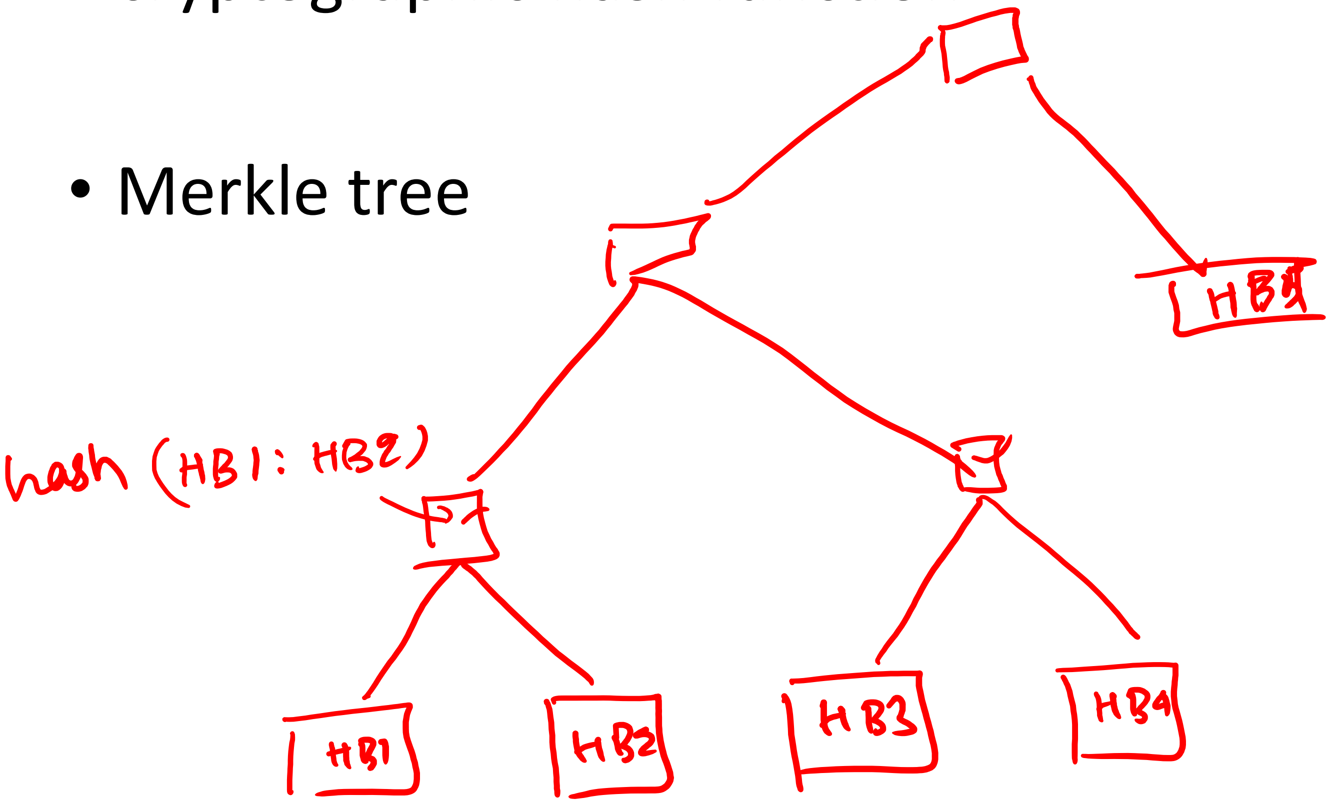
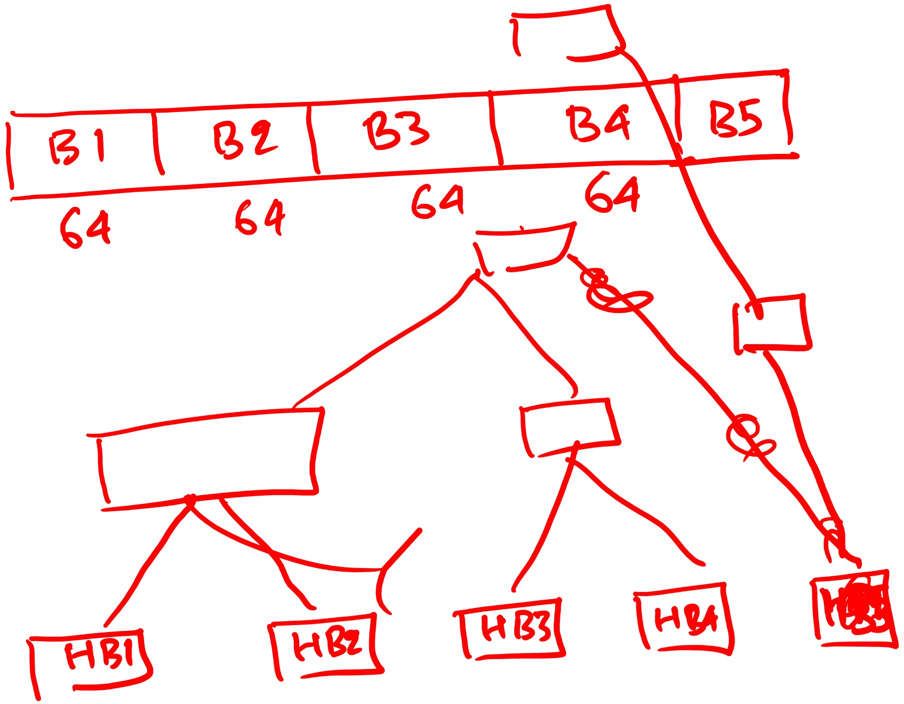
Integrity of file system in Linux

- cryptographic hash function
- Merkle tree





Shared memory IPC in xv6

- Implement IPC between parent and child
- An application can do at most one fork
- The parent process is always a sender
- The child is always a receiver

Shared memory IPC in xv6

- Implement a system call to initiate shared memory IPC
- Map a single page (shared page) in the process address space for IPC
- Always use a fixed address for the shared page
  - e.g., KERNBASE – 4096
- In fork system call, use the same physical page at (KERNBASE-4096) as the parent process

Shared memory IPC in x86

- Implement send and receive routine in the user library (ulib.c)
- The send and receive routine can assume that the shared page is present at (KERNBASE-4096)
- Implement a circular queue in send and receive

User mode library (ulib.c)

```
struct queue {
    int head, tail;
    char buf[4096-8];
};
// declare a global variable in ulib.c
struct queue *ipc_queue = (struct queue*)(KERNBASE-4096);
```

- Use ipc_queue in send and receive implementation

User mode library

- add send and receive routines to ulib.c
- declare the prototype of send and receive routines in user.h
- add a test case similar to memtest1 (from assignment 3) that uses the send and receive interfaces
  - search for memtest1 in the Makefile of the assignment3 folder

System call handler

- Add a new system call in syscall.c, syscall.h, sysproc.c, user.h, usys.S

- define a new system call vector in syscall.h
- add a handler corresponding to vector in syscall.c
- define the handler in sysproc.c
- define the user mode system call routine in usys.S
- declare the prototype of user mode routine in user.h

Scheduling

- FIFO
- round-robin
- priority scheduling
- priority donation

How a mature OS implement scheduling?

- Multilevel feedback queue scheduler
  - maintains multiple queues
  - each queue has a different priority
  - I/O intensive and interactive workloads are given highest priority
  - compute-intensive applications have lower priority
  - priority of low priority process is increased over time if it is not getting scheduled

Multiple queue feedback scheduling

How to identify an I/O intensive application

- An I/O intensive workload is likely to yield early
- A high priority thread is given smaller time quantum
- If a thread yields before exhausting the time quantum then the application is considered as I/O intensive
- If a thread exhausts its time quantum, then it is moved to the lower priority queue with larger time quantum

Multilevel feedback queue scheduling

- Parameters used for multilevel feedback queue scheduler
  - The number of queues
  - The scheduling algorithm for each queue
  - When to move a process to a lower priority queue
  - When to move a process to a higher priority queue
  - Which queue should be assigned initially

Multilevel feedback queue scheduling (example)

0	1 ms
1	2 ms
2	4 ms
3	8 ms
4	16 ms

- Number of queues are 8
- priority (queue[i]) > priority (queue[j]), where i < j
- time_quantum(queue[i+1]) = 2 * time_quantum(queue[i])
- A new thread always added to queue[0]

Multilevel feedback queue scheduling (example)

- If a thread in queue[i] yields before quantum(queue[i]), then it is moved to queue[i-1] (if exists)

- If a thread in queue[i] yields after quantum(queue[i]), then it is moved to queue[i+1] (if exists)

- If a thread in queue[i] was not scheduled in last “x” time quantum move it to queue[i-1] (if exists)

- Individual queue implements round-robin scheduling

Multiple-Processor scheduling

- Processor affinity
  - most multiprocessor OS try to schedule a process on the same processor every time
    - to take the full advantage of per-CPU cache
  - However, this is not possible all the time (for example, consider a case when a CPU is idle, but a process is waiting to be scheduled on other CPU)
- Linux provides system calls to set the processor affinity to a set of cores
  - In this case, a process or thread will always be scheduled on a set of CPUs

What is inside an OS?

- scheduler
- exception handlers
- system call handlers
- device drivers
- file system
- network stack (TCP/IP)
- accelerators (GPU)
- and so on

How does an OS provide process isolation?

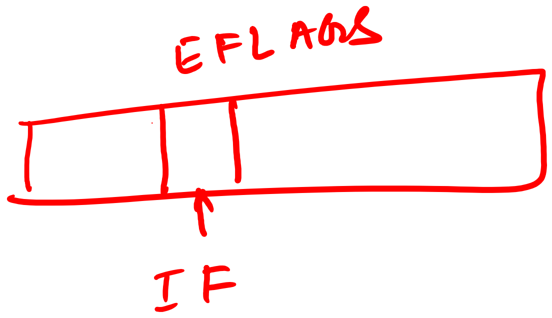
- Memory isolation
  - Using page tables
- Disallow execution of privileged instructions
  - Using protection rings

Do we need protection rings if the compiler is trusted?

- Let us assume that the compiler never generates a privileged instruction

Do we need protection rings if the compiler is trusted?

- Some instructions have different meanings in different protection rings
  - pushf and popf instructions save and restore the eflags registers on/from the stack
  - in ring 0, popf restores the interrupt flag from the stack



Do we need protection rings if the compiler is trusted?

- Can you execute arbitrary byte code in a C program

Do we need protection rings if the compiler is trusted?

```
void foo () {
    ...
}
int main () {
    void (*fnptr)() = foo;
    ...
    fnptr ();
    return 0;
}
```

fnptr++;

ptr = x

0x1000

jmp ~~...~~ 0x1000

foo:

b8 fa 00 00 00 mov $0xfa, %eax

~~fa ... cli~~

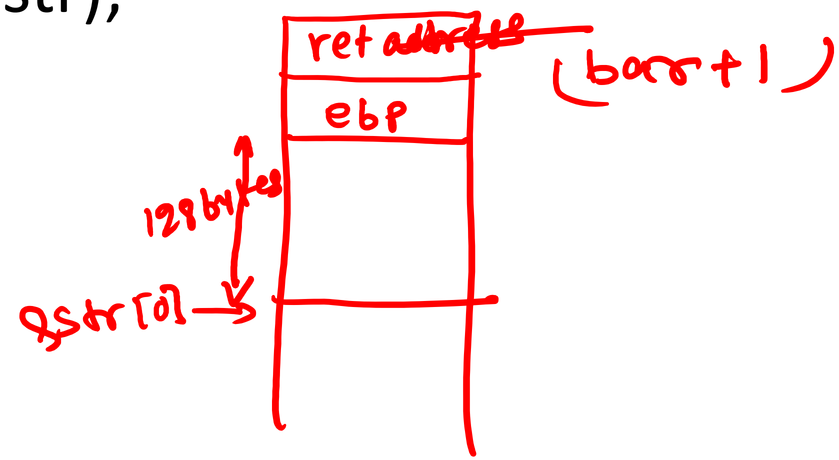
Do we need protection rings if the compiler is trusted?

```
foo () {
    char str[128];
    scanf ("%s", str);
    …
}
```

bar:

b8 fa 00 00 00   **mov $0xfa, %eax**

~~**fa**~~   ~~**cli**~~



Next class

- How can we build an OS which offers better security?