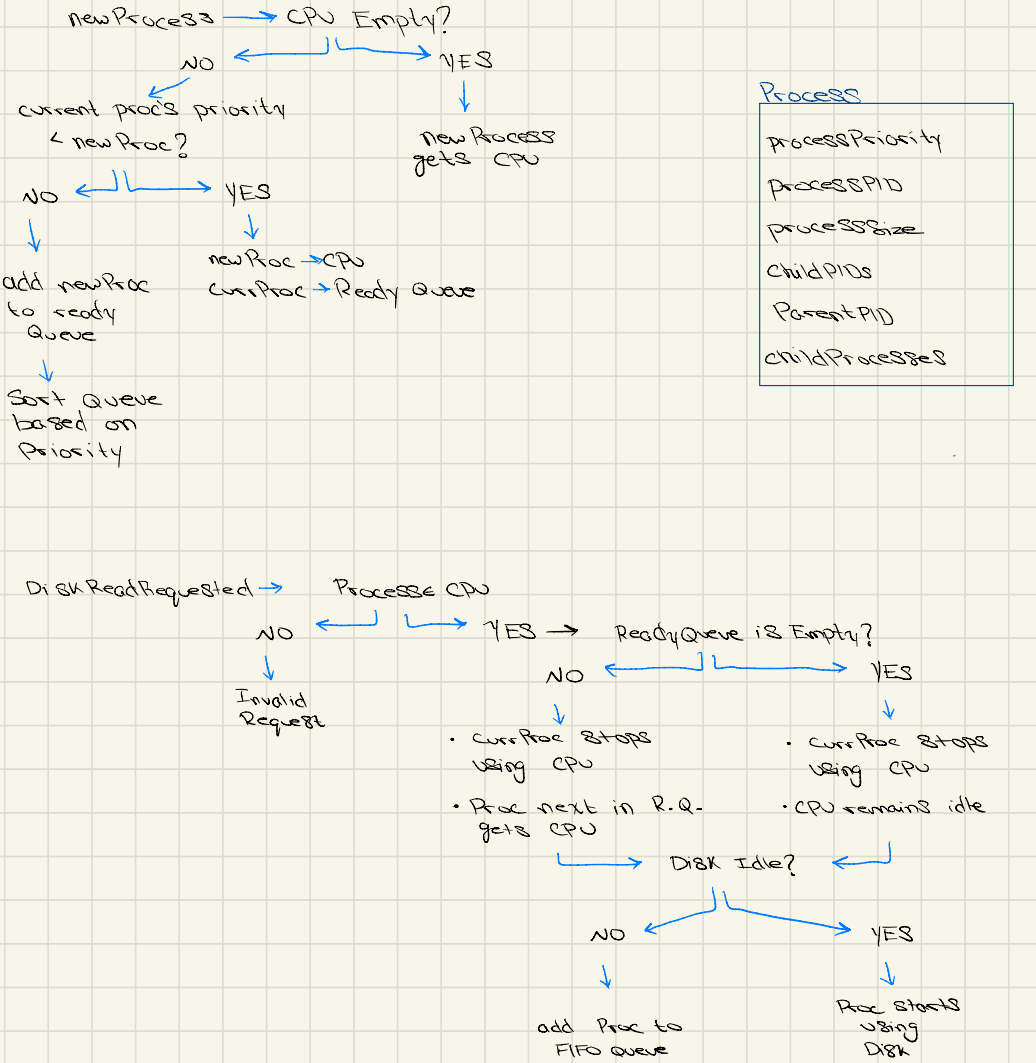

newProcess
CPU Empty?
NO
YES
current proc's priority < newProc?
new Process gets CPU
NO
YES
add newProc to ready Queue
newProc → CPU
CurrProc → Ready Queue
Sort Queue based on Priority
Process
processPriority
processPID
processSize
childPIDs
ParentPID
childProcesses
DiskReadRequested
Processe CPU
NO
YES
ReadyQueue is Empty?
Invalid Request
NO
YES
• CurrProc stops using CPU
• Proc next in R.Q. gets CPU
• CurrProc stops using CPU
• CPU remains idle
Disk Idle?
NO
YES
add Proc to FIFO queue
Proc starts using Disk
DiskJobComplete
Exit Disk
DiskQueue Empty?
CPU Idle?
*get info of returning proc before
YES
NO
Set disk to idle state
NO
YES
Proc gets CPU
Next Proc Start using disk
RetProcPri > RunProPri?
Set 1st process waiting as Proc using disk & remove from disk Queue
NO
YES
RunPro → RQ
RetPro → CPU
RetPro → RQ
Sort RQ

NextProcessUP() → ReadyQueue Empty?

NO
- Process ∈ ReadyQueue[0] ⇒ next proc to Run
- remove ReadyQueue[0] from Queue

YES
- set CPU to idle ⇒ PID = 0, Proc = DummyProc

GetDisk(x) → HD #x exists?
- NO → Invalid Req
- → get PID of Process using Disk + file it's reading } Package into FileReadRequest obj & Return

GetReadyQueue() → iterate over ReadyQueue → For each proc get ProcessPID & place into arr to be returned

RAM:

represented as vector<MemoryItem>

amountOfRam = LIMIT - Process' sizes summed up can't go above this##

Keep track - currRamSize

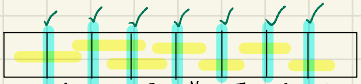
NewProcess → newProcSize + currRamSize > amountOfRam?
- NO → can't create process
- YES → scan over RAM vector to find 1st hole large enough

vec[3].addr - (vec[2].addr + vec[2].size) >= newProcess.size?
- NO → keep searching
  - no hole large enough? (reached end of vec) → can't create process
- YES → create process (get PID)
  - copy elements up until vec[2]
  - insert newProcess @ vec[3]
  - copy remaining old vec
  - currRamSize = newProc.size

addr + size

Pseudo code →

Scanning All Possible Holes




0: hole between 0 & 1
1: 1 & 2
2: 2 & 3
3: 3 & 4
4: 4 & 5
5: 5 & 6
6: 6 & 7
7:

Search until i < Size-1

↑ Implement Best-Fit Algo/Approach

holeSpace + usedSpace = totalRam

Remove Current CPU Process from Memory

↓

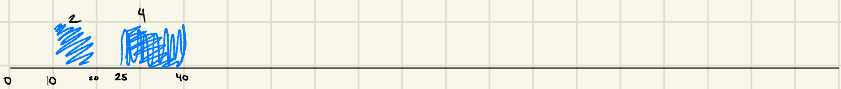
Scan vector until you Find vec[x].PID = currProcPID

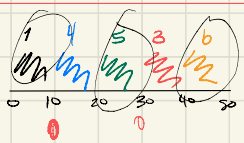
↓

RAM.erase(RAM.begin()+x)

Scratch work for RAM calculations

CPU = 4




10 - (0+10) = 0 > 5 X

(30) - (10+10) = 10 > 8

fork() → Create childProcess → maintain link → Parent keeps track of its child Processes

↓

Child keeps track of it's Parent Processes

SimWait(): Process exits CPU → Place into waitingProc vector

SimExit() → Process has Parent?

NO → process has children?

NO → terminate process

YES → terminate decendents

YES → Parent ∈ waitingProc?

YES → Remove Parent ∈ waitingProc → Parent → RQ or CPU

NO → turn into zombieProc → terminate Process

cascading termination (Scroll Down)

- ***void SimWait()***
  The process wants to pause and wait for any of its child processes to terminate. Once the wait is over, the process goes to the end of the ready-queue or the CPU. If the zombie-child already exists, the process proceeds right away (keeps using the CPU) and the zombie-child disappears. If more then one zombie-child exists, the system uses one of them (any!) to immediately restart the parent and other zombies keep waiting for the next wait from the parent.

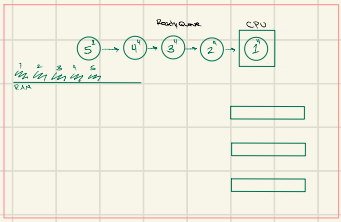
- ***void SimExit()***
  The process that is currently using the CPU. Make sure you release the memory used by this process immediately. If its parent is already waiting, the process terminates immediately and the parent becomes runnable (goes to the ready-queue or CPU). If its parent hasn't called wait yet, the process turns into zombie.
  To avoid the appearance of the orphans, the system implements the cascading termination. Cascading termination means that if a process terminates, all its descendants terminate with it.

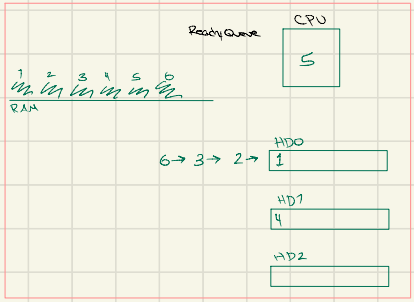
- ***void SimWait()***

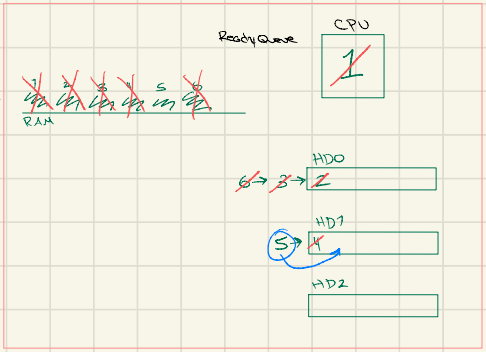
# Cascading Termination:

```
SimOS sim{ 3, 10'000'000'000 };

sim.NewProcess(4, 10);
sim.SimFork();
sim.SimFork();
sim.SimFork();
sim.NewProcess(1, 10);

sim.DiskReadRequest(0, "abc");
sim.DiskReadRequest(0, "abc");
sim.DiskReadRequest(0, "abc");
sim.SimFork();
sim.DiskReadRequest(1, "abc");
sim.DiskReadRequest(0, "abc");

CHECK(sim.GetCPU() == 5);
sim.DiskReadRequest(1, "abc");

sim.DiskJobCompleted(0);
sim.SimExit();

CHECK(sim.GetCPU() == 0);
auto q1{ sim.GetDiskQueue(1) };
CHECK(q1.size() == 0);

CHECK(sim.GetDisk(0).PID == 0);
CHECK(sim.GetDisk(1).PID == 5);
CHECK(sim.GetDiskQueue(0).size() == 0);
CHECK(sim.GetDiskQueue(1).size() == 0);

auto mem{ sim.GetMemory() };
CHECK(mem.size() == 1);
CHECK(ContMemoryItem(mem[0], 40, 10, 5));

return TEST_PASS;
```

↑ Test case




Test Case Visualization

For children of Process to Exit:
- if child ∈ ReadyQueue
- if child ∈ Disk
- if child ∈ DiskQueue
- if child ∈ RAM
- if child ∈ ZombieProcess

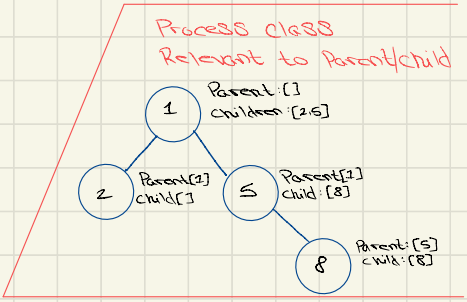
} Terminate Child

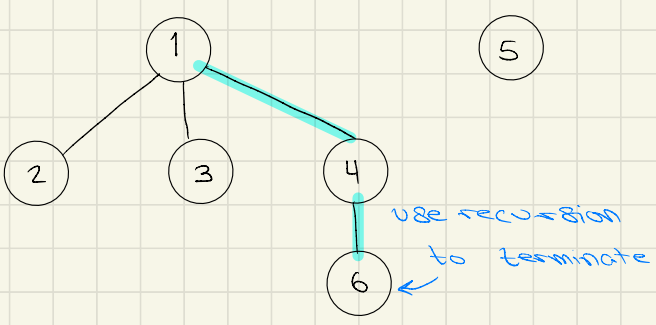
← does NOT Account for grandchildren

Solution Below

Kill process ∈ CPU
load next process

} Terminate Parent



```
RemoveDecendentsOf(ParentPID)
  scan RQ: if procERQ's parentPID == parentPID
      RemoveDecendentsOf(procPID)
      erase
  scan DQ: if procEDQ's parentPID == parentPID
      RemoveDecendentsOf(procPID)
      erase
  check Disk: if procEDisk's parentPID == parentPID
      RemoveDecendentsOf(procPID)
      remove
  scan RAM: if procERAM's parentPID == parentPID
      RemoveDecendentsOf(procPID)
      remove
```

FAILED APPROACH ::

hard to access RAM + keep track of parentPIDs

```
for ith child ∈ children:
  if childPIDeRQ → erase
  if childPIDe Disk → Remove from disk
  if childPIDe DiskQueue → erase
  if childPIDe RAM → erase
  if childPIDe zombieProc → erase
  terminate(child[i])
```

↑ parent already !!
can't access child

FAILED Approach ⇒ does NOT Account for grandchildren

## Solution to Cascading Termination (Recursive):

```
TerminateChild(Process ParentProcess){
    children = parentProcess.getChildrenProcesses();
    For each child ∈ children:
        Scan ReadyQueue → if process ∈ R.Q. == child
                              => TerminateChild(child)
                                 remove process from R.Q.
        Scan each HardDisk → if process ∈ DiskQueue == child
                              => TerminateChild(child)
                                 remove process from D.Q.
        Scan each HardDisk → if runningProcess ∈ H.D. == child
                              => TerminateChild(child)
                                 end runningProcess ∈ H.D.
        Scan RAM → if MemoryItem ∈ RAM belongs to child
                   => Erase item from RAM
}
TerminateChild(Parent)
DeleteRunningProcess()   Remove parent from CPU + RAM
```