2024_final_report

1. Trace Code

1-1 New → Ready

*1-1. New→**Ready***

- userprog/userkernel.cc UserProgKernel::InitializeAllThreads()
- userprog/userkernel.cc UserProgKernel:: InitializeOneThread(char*, int, int)
- threads/thread.cc Thread::Fork(VoidFunctionPtr, void*)
- threads/thread.cc Thread::StackAllocate(VoidFunctionPtr, void*)
- threads/scheduler.cc Scheduler::ReadyToRun(Thread*)

- `UserProgKernel::InitializeAllThreads()`
    - 設定thread初始化資訊，傳入`InitializeOneThread()`

- `UserProgKernel::InitializeOneThread(char*, int, int)`
    - 為新的thread分配記憶體空間，並且fork the thread
    - 先呼叫`ForkExecute()`，於memory載入program file及初始化一些變數後，將結果傳入`Fork()`
    - 記錄thread數量的變數threadNum+1

- `Thread::Fork(VoidFunctionPtr, void*)`
    - 為fork的thread建立interrupt、scheduler、oldLevel變數

    - 呼叫StackAllocate() ⇒ 分配執行時會用到的stack空間
    - 改變IntStatus至IntOFF，關閉interrupt發送
    - 呼叫ReadyToRun() ⇒ 將thread放進ready queue
    - 打開interrupt

- Thread::StackAllocate(VoidFunctionPtr, void*)
    - 於memory中分配空間，作為給新的forked thread的execution stack
    - x86:傳入SWITCH()的位址需要是ThreadRoot位址

- Scheduler::ReadyToRun(Thread*)
    - 確認interrupt status是IntOFF(關閉狀態)
    - thread status設定為READY
    - 將thread放入ready queue

1-2 Running → Ready

1-2. Running→Ready

machine/mipssim.cc	Machine::Run()
machine/interrupt.cc	Interrupt::OneTick()
threads/thread.cc	Thread::Yield()
threads/scheduler.cc	Scheduler::FindNextToRun()
threads/scheduler.cc	Scheduler::ReadyToRun(Thread*)
threads/scheduler.cc	Scheduler::Run(Thread*, bool)

- 遇到interrupt會將thread從Running切換到Ready，等待interrupt處理好再繼續執行
- Machine::Run()
    - 模擬user level program執行
    - 建立instruction pointer，儲存decode instruction
    - 切換至user mode

    - 呼叫OneInstruction()，執行user-level program的instruction的需求
    - 呼叫OneTick()，將模擬時間計時器調快，以及檢查是否有在等待的interrupt要執行

- Interrupt::OneTick()
    - 當user instruction executed或是interrupt re-enabled時會呼叫OneTick()
    - 將模擬時間計時器調快(advanced stimulated time)
    - ChangeLevel()關閉interrupt
    - CheckIfDue()檢查是否還有interrupt準備好要被發送
    - ChangeLevel()開啟interrupt
    - 設定status為SystemMode，其代表kernel mode，因為接下來要呼叫的Yield()屬於kernel routine
    - 呼叫Yield()

- Thread::Yield()
    - 首先儲存舊的status，並且把interrupt status設定為IntOFF(disabled)
    - 把目前在執行的這個thread暫停，讓其他準備好的thread先執行
    - 尋找是否有下一個要執行的thread(next thread)
    - 如果有，則將目前在執行的這個thread放進ready list
    - 呼叫Run，讓下一個thread(next thread)先執行
    - 最後把interrupt status設定為IntOFF(disabled)

- Scheduler::FindNextToRun()
    - 先確認目前interrupt status是IntOFF(disabled)，這部分在Yield()已經先設定好
    - 檢查ready list是否有thread在等待：

        - 沒 有 → return NULL
        - 有 → 移除存在ready list的第一個thread，並return被移除的thread，也就是next thread to run

- Scheduler::ReadyToRun(Thread*)
    - 檢查目前interrupt status是IntOFF(disabled)
    - 根據priority將thread放進ready queue排隊

- Scheduler::Run(Thread*, bool)
    - 新舊thread的交換，儲存舊的thread，load並執行新的thread
    - 檢查interrupt status為IntOFF(disabled)
    - 檢查舊的thread是否已經finish(要被刪除)，若有，則將該thread存在toBeDestroyed
    - #ifdef USER_PROGRAM：若舊的thread是user program生成的，則將user’s CPU register的資料存起來
    - 檢查舊的thread是否有stack overflow
    - 改變current thread(new thread) status為RUNNING
    - SWITCH() ⇒ context switch，新舊thread的register資料交換存取
    - CheckToBeDestroyed() ⇒ 刪除已經完成的old thread
    - #ifdef USER_PROGRAM：若刪除了thread需要復原空間，則恢復其狀態

1-3 Running → Waiting

1-3. Running→Waiting (Hint: When a thread has a console output(I/O), it needs to yield CPU resource and go to waiting state.)

- userprog/exception.cc ExceptionHandler(ExceptionType) case SC_PrintInt
- userprog/synchconsole.cc SynchConsoleOutput::PutInt()
- machine/console.cc ConsoleOutput::PutChar(char)
- threads/synch.cc Semaphore::P()
- threads/synchlist.cc SynchList<T>::Append(T)
- threads/thread.cc Thread::Sleep(bool)
- threads/scheduler.cc Scheduler::FindNextToRun()
- threads/scheduler.cc Scheduler::Run(Thread*, bool)

- 以console output為範例：當發生I/O interrupt時，需要yield CPU資源，走running → waiting的過程
- ExceptionHandler(ExceptionType) case SC_PrintInt
    - 從register 4取value(val)
    - 呼叫System call PrintInt (模擬I/O狀況)
    - 呼叫PutInt()

- SynchConsoleOutput::PutInt()
    - 將要顯示在console display的value存放在char array
    - lock->Acquire() ⇒ 鎖住thread並等待process unlock，避免console output時被其他程式打斷
    - 呼叫PutChar()
    - 呼叫Semaphore::P()
    - lock->Release() ⇒ 解除先前的鎖定

- ConsoleOutput::PutChar(char)
    - 印出字元，之後發出interrupt
    - putBusy ⇒ 因為同時不能有2個以上的thread執行PutChar()，故設定為TRUE，使同時間之下只有一個thread執行

- Semaphore::P()
    - 因為不能被打斷interrupt status設定為IntOFF
    - 若semaphore value = 0，表示還不能執行，於是將thread先放在一旁(放進queue排隊)，並將他暫停執行(呼叫Sleep())
    - 若semaphore value > 0，表示可以執行，並則將此value遞減
    - 將interrupt改回enable

- SynchList<T>::Append(T)
    - 啟用mutual exclusive lock，確保thread新增至queue時不會被中斷

    - 將正在等待的current thread放進queue的尾端
    - 結束後呼叫Signal()讓其他等待者可以準備執行
    - 解除lock

- Thread::Sleep(bool)
    - 由於current thread在等待semaphore value達到一個目標值(> 0)，所以用Sleep()，先將此thread放在一旁等待(status = BLOCKED)
    - 等時機到的時候，再由其他thread喚醒排在queue裡面、還在等待的current thread
    - 而當current thread還在等待時，呼叫FindNextToRun() ⇒ 先找下一個準備好的thread來執行
        - 如果也沒有下一個thread等待執行，則進入Idle()的狀態
    - 更新Remaining Burst Time、重設current_thread、執行context switch
    - 呼叫Run()執行thread

- Scheduler::FindNextToRun()
    - 尋找ready list中，下一個要執行的thread
    - 若無，則回傳NULL

- Scheduler::Run(Thread*, bool)
    - finishing = FALSE ⇒ 表示還沒完成執行，current thread不會被丟進toBeDestroyed
    - 若current thread是user program，儲存register data
    - 讓next thread當家，以及設定其status為RUNNING
    - SWITCH()進行current和next thread的context switch
    - 檢查interrupt state是IntOFF(從SWITCH回來會是IntOFF)
    - 檢查並刪除toBeDestroyed，但是這裡current thread還沒執行結束所以不會被刪除

- 總之，current thread現在處於BLOCKED狀態，等待console output的I/O event結束再重回執行

1-4 Waiting → Ready

1-4. Waiting→Ready (Hint: After finishing console output(I/O), this thread can return to ready queue.)

- threads/synch.cc　　Semaphore::V()
- threads/scheduler.cc　　Scheduler::ReadyToRun(Thread*)

- Semaphore::V()
  - 設定interrupt status為IntOFF
  - 如果queue不是空的(queue->IsEmpty() = 0)，也就是有還在等待的thread在queue裡，則將thread傳入ReadyToRun()
  - 增加semaphore value
  - interrupt status設定回原本的enable

- Scheduler::ReadyToRun(Thread*)
  - 設定狀態為READY
  - 放入ready list，等待被讀取與執行

1-5 Running → Terminated

1-5. Running→Terminated (Note: start from the Exit system call is called)

- userprog/exception.cc　　ExceptionHandler(ExceptionType) case SC_Exit
- threads/thread.cc　　Thread::Finish()
- threads/thread.cc　　Thread::Sleep(bool)
- threads/scheduler.cc　　Scheduler::FindNextToRun()
- threads/scheduler.cc　　Scheduler::Run(Thread*, bool)

- Exit the thread process
- ExceptionHandler(ExceptionType) case SC_Exit
  - 呼叫System call Exit，終止thread

- 呼叫Finish()

- Thread::Finish()
  - 設定interrupt status為IntOFF
  - 檢查當下的thread是current thread(避免終止到別的thread)
  - 呼叫Sleep()，傳入TRUE表示此thread已經是finish狀態

- Thread::Sleep(bool)
  - 因為已經執行完畢的thread需要由下一個thread來刪除，所以先將此thread放在一旁等待(status = BLOCKED)
  - 找ready list裡的下一個thread，呼叫FindNextToRun() ⇒ 先找下一個準備好的thread來執行
    - 如果也沒有下一個thread等待執行，則進入Idle()的狀態
  - 將next thread和TRUE傳入Run()，準備交換current和next

- Scheduler::FindNextToRun()
  - 尋找下一個thread
  - 若無，則回傳NULL

- Scheduler::Run(Thread*, bool)
  - 因為參數finishing是TRUE，所以執行完畢的thread會被存入toBeDestroyed
  - 等到current和next thread狀態交換完畢，呼叫CheckToBeDestroyed()，刪除toBeDestroyed裡的thread
  - 結束此thread

1-6 Ready → Running

1-6. Ready→Running

- threads/scheduler.cc Scheduler::FindNextToRun()
- threads/scheduler.cc Scheduler::Run(Thread*, bool)
- threads/switch.s SWITCH(Thread*, Thread*)
- Machine/mipssim.cc for loop in Machine::Run()

- scheduler dispatch
- Scheduler::FindNextToRun()
    - 尋找下一個要執行的thread
    - 接著進入Run()
- Scheduler::Run(Thread*, bool)
    - 要讓next thread(剛剛從ready list取出的thread)執行前，需要交換current thread的狀態
    - SWITCH()以前的function在前面已有敘述
    - 呼叫SWITCH()，進行register data的變動
- SWITCH(Thread*, Thread*)
    - context switch：儲存前一個thread的狀態資訊，並更新接下來要執行的thread資訊
    - 語法：movl [source] [destination] ⇒ 將source搬到destination，movl是指move long
    - 儲存thread 1資訊到memory
        - 將eax的值存到_eax_save
        - 把thread 1(t1)的指標存到eax
        - 接著將其他registers的值存到以eax的位址開始、+4的倍數的位址們(t1的相關資訊)
        - 取回先前存的eax的值，存到t1的eax+4
        - 取得return address，存到t1的eax+32

- 把thread 2資訊從memory更新到CPU register
    - 把thread 2(t2)的指標存到eax
    - 藉由ebx，把新的eax值存到_eax_save
    - 接著將以eax的位址開始、+4的倍數的位址們存到其他registers的值(t2的相關資訊)
    - 將return adress存到t2的eax+32，再將此address更新到t2指標
    - 把先前存好的_eax_save的值更新給eax
- return
- 完成舊的thread 1以及新的thread 2的資訊交換

- for loop in Machine::Run()
    - 讀取next thread的instructions並執行

2. Implementation

2-1 Multilevel feedback queue scheduler

- userprog/userkernel.cc

  In InitializeOneThread(), after construct a Thread, store the priority of it.

```
{
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->SetPriority(pr);
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, /
     (void *)t[threadNum]);
    threadNum++;

    return threadNum-1;
}
```

  In ForkExecute(),

```
{
    if ( !t->space->Load(t->getName()) ) {
        return;             // executable not found
    }

    t->space->Execute(t->getName());
}
```

- threads/thread.h

  In Thread(), we need to add some functions

```
{
    int GetExecTick();
    int GetLayer();
    int GetPriority() { return priority; }
    void SetPriority(int expected) /
    { priority = expected; }
    void SetInitialTick(int tick) /
    { initialTick = tick; }
    int AccumulatePriority(int addPriority);
    void SetAgeInitialTick(int expected) /
    { initialAgeTick = expected; }
    int GetIsExceedAgeTime() /
    { return totalAge >= 400; }
    void UpgradeTotalAgeTick();
    void DecreaseTotalAge(int decreaseTick) /
    { totalAge -= decreaseTick; }
    int GetTotalAge() { return totalAge; }
    double GetApproximateBurstTime();
    int RecalculateBurstTime();
    void TerminateBurstTimeCounting();
      int getID() { return (ID); }
}
```

- threads/thread.cc

In Thread(), we add some arguments.

```
{
    priority = 0;
    initialTick = 0;
    burstTime = 0.0;
    predictTime = 0.0;
    lastExecTime = 0.0;
    initialAgeTick = 0;
    totalAge = 0;
}
```

Implement the functions we added in thread.h as below:

```
int Thread::GetExecTick() {
    // the line below means it has terminate
    // (in waiting state),
    // we grab the last execute result.
    if (burstTime <= 0) return lastExecTime;
    // the line below means it just stop
    // (in ready state),
    // we grab the current burst time.
    else return burstTime;
}

// When Thread::Sleep(), running -> waiting
void Thread::TerminateBurstTimeCounting() {
    // Confirm burst time (execution time) first.
    RecalculateBurstTime();

    double newPredictTime = /
    (double)(predictTime/2) + (double)(burstTime/2);
    DEBUG(dbgExpr, /
    "[D] Tick ["<< kernel->stats->totalTicks <<"]: /
    Thread [" << ID << "] update approximate burst time,/
    from: ["<< predictTime <<"], add ["<< burstTime <<"],/
    to ["<< newPredictTime <<"]");
```

```
    predictTime = newPredictTime;
    lastExecTime = burstTime;
    burstTime = 0.0;
}

// When Thread::Yield(), running -> ready
// OR When Thread::Sleep(), running -> waiting
int Thread::RecalculateBurstTime () {
    burstTime += /
    (double)(kernel->stats->totalTicks - initialTick);
    return burstTime;
}


double Thread::GetApproximateBurstTime() {
    // According to Discuss Room,
    // we don't need to check for
    // "remaining approximate burst time",
    // "approximate burst time" is good enough
    return predictTime;
}

int Thread::AccumulatePriority(int addPriority)
    { if (priority + addPriority <= 149) /
    priority += addPriority;
    else priority = 149;
    return priority;
}

void Thread::UpgradeTotalAgeTick() {
    // Why direct add 100 ?
    // 1. Prevent when first clock comes,
    //    this thread is not enough for 100.
    // 2. Also useful when it's going to run,
    //    we can add remaining tick
    //    from last initial tick back to total age
    // If above 149, stop upgrade
    if (priority >= 149) return;
```

```
    totalAge += kernel->stats->totalTicks - initialAgeTick
}

int Thread::GetLayer() {
    if (priority >= 100) return 1;
    else if (priority >= 50 && priority <= 99) return 2;
    else return 3;
}
```

In Yield(),we put current thread into queue to compare and find the next thread to run, then confirm current thread burst time.

```
{
    // We should put current thread into queue,
    // in order to compare with thread in queue.
    kernel->scheduler->ReadyToRun(this);
    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) {
        // Confirm current thread burst time
        RecalculateBurstTime();
        kernel->scheduler->Run(nextThread, FALSE);
    }
}
```

In Sleep(), let oldThread start to calculate burst time.

```
{
    status = BLOCKED;
    TerminateBurstTimeCounting();
}
```

- threads/scheduler.cc

In scheduler, we replace a single "readylist" with 3 lists, L1, L2 and L3.

```
Scheduler::Scheduler()
{
    L1 = new List<Thread *>;
    L2 = new List<Thread *>;
    L3 = new List<Thread *>;
    toBeDestroyed = NULL;
}

Scheduler::~Scheduler()
{
    delete L1;
    delete L2;
    delete L3;
}
```

Based on processe's current priority, save them in different queues.

```
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " /
    << thread->getName());
    //cout << "Putting thread on ready list: " << thread->
    thread->setStatus(READY);

    int currentPriority = thread->GetPriority();
    if (currentPriority >= 100) {
        PutIntoQueue(1, L1, thread);
    } else if (currentPriority >= 50 /
    && currentPriority <= 99) {
        PutIntoQueue(2, L2, thread);
    } else {
        PutIntoQueue(3, L3, thread);
```

```
    }
    thread->SetAgeInitialTick(kernel->stats->totalTicks);
}
```

We implement FindNextToRun() according to different scheduling algos in L1, L2 and L3.

```
Scheduler::FindNextToRun ()
{
    // DEBUG(dbgExpr, "[X] FindNextToRun");
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    /// shareable variable
    Thread *iterThread;
    ListIterator<Thread *> *iter;
    ///
    if (!L1->IsEmpty()) {
        // Preemptive SJF
        // Assign default value (The first element)
        // to prevent segmentation fault
        Thread *approximateThread = L1->Front();
        iter = new ListIterator<Thread *>(L1);
        for (; !iter->IsDone(); iter->Next()) {
            iterThread = iter->Item();
            if (iterThread->GetApproximateBurstTime() < /
            approximateThread->GetApproximateBurstTime())
            {
                approximateThread = iterThread;
            }
        }
        return RemoveFromQueue(1, L1, approximateThread);
    } else if (!L2->IsEmpty()) {
        // Non-preemptive priority
        Thread *highestPriorityThread = L2->Front();
        iter = new ListIterator<Thread *>(L2);
        for (; !iter->IsDone(); iter->Next())
            { iterThread = iter->Item();
```

```
            if (iterThread->GetPriority() > /
            highestPriorityThread->GetPriority()) {
                highestPriorityThread = iterThread;
            }
        }
        return RemoveFromQueue(2,L2,highestPriorityThread)
    } else if (!L3->IsEmpty()) {
        // Round-robin
        return RemoveFromQueue(3, L3, L3->Front());
    } else {
        return NULL;
    }

}
```

```
Thread* Scheduler::PutIntoQueue(int layerIdx, /
List<Thread *> *cacheList, Thread *newThread) {
    cacheList->Append(newThread);
    DEBUG(dbgExpr, /
    "[A] Tick ["<< kernel->stats->totalTicks <<"]: /
    Thread [" << newThread->getID() << "] is inserted /
    into queue L["<< layerIdx <<"]");
    // If L1
    //if (layerIdx == 1) PreemptiveCheck(newThread);
    return newThread;
}

Thread* Scheduler::RemoveFromQueue(int layerIdx, /
List<Thread *> *cacheList, Thread *newThread) {
    cacheList->Remove(newThread);
    DEBUG(dbgExpr, /
    "[B] Tick ["<< kernel->stats->totalTicks <<"]: /
    Thread [" << newThread->getID() << "] is removed /
    from queue L["<< layerIdx <<"]");
    // Calculate remaining tick from last check point,
    // and add back to thread's total age.
    newThread->UpgradeTotalAgeTick();
    // Keep current tick data to thread struct,
```

```
    // it's useful when this thread is transfered
    // in aging rather than go to execute.
    newThread->SetAgeInitialTick(kernel->stats->totalTicks
    return newThread;
}
```

Then, we add the aging function.

```
void Scheduler::AgingProcess()
    { PerAgingProcess(L1, 1);
    PerAgingProcess(L2, 2);
    PerAgingProcess(L3, 3);
}
```

When a process has been waited over 400 ticks, its priority increased 10. Then we check its new priority to determine whether it should go to the upper queue.

```
void Scheduler::PerAgingProcess( /
List<Thread *> *cacheList, int currentLayer)
    { ListIterator<Thread *> *iter;
    Thread *iterThread;
    iter = new ListIterator<Thread *>(cacheList);
    for (; !iter->IsDone(); iter->Next()) {
        iterThread = iter->Item();
        int foundPriority = iterThread->GetPriority();
        // Add 100 to thread's total age tick ()
        iterThread->UpgradeTotalAgeTick();
        // Keep current tick data to thread struct,
        // it's useful when this thread is transfered to
        // running state.
        iterThread->SetAgeInitialTick( /
        kernel->stats->totalTicks);
        // Whether this thread total waiting tick is
        // above 400
        bool isExceedAgeTime = /
```

```
        iterThread->GetIsExceedAgeTime();
        bool canStillAddPriority = foundPriority < 149;
        if (isExceedAgeTime && canStillAddPriority) {
            iterThread->DecreaseTotalAge(400);
            iterThread->AccumulatePriority(10);
            DEBUG(dbgExpr, /
            "[C] Tick ["<< kernel->stats->totalTicks <<"]:
            Thread [" << iterThread->getID() << "] changes
            its priority from ["<< foundPriority <<"] to /
            ["<< iterThread->GetPriority() <<"]");
            // Manage L3->L2 L2->L1
            if (currentLayer == 3 /
            && iterThread->GetPriority() >= 50)
                { RemoveFromQueue(3, L3,
                iterThread); PutIntoQueue(2, L2,
                iterThread);
            } else if (currentLayer == 2 /
            && iterThread->GetPriority() >= 100)
                { RemoveFromQueue(2, L2,
                iterThread); PutIntoQueue(1, L1,
                iterThread);
            }
        }
    }
}
```

- threads/scheduler.h

  After finishing scheduler.cc, we also need to declare the functions in scheduler.h

```
class Scheduler(){
    bool hasThreadInL1() { return !(L1->IsEmpty()); }
    void AgingProcess();
    void PerAgingProcess(List<Thread *> *cacheList, /
    int currentLayer);
    void PreemptiveCheck(Thread *newThread);
    Thread* PutIntoQueue(int layerIdx, /
    List<Thread *> *cacheList, Thread *newThread);
    Thread* RemoveFromQueue(int layerIdx, /
```

```
    List<Thread *> *cacheList, Thread *newThread);
}
```

- threads/alarm.cc

  Do the aging process every 400. Alarm does callback every 100 ticks, so we call aging process when call back.

```
Alarm::CallBack(){
    kernel->scheduler->AgingProcess();
    if (status == IdleMode) {
        interrupt->YieldOnReturn();
    }
}
```

2-2 Command line argument -epb

- userprog/userkernel.cc

  In UserProgKernel(),implement -epb line argument.

  There are 4 arguements after "-epb": 2 processes、priority、burst time.

```
else if (strcmp(argv[i], "-epb") == 0) {
            execfile[++execfileNum] = argv[++i];
            threadPriority[execfileNum] = /
            atoi(argv[++i]);
            threadRemainingBurstTime[execfileNum] = /
            atoi(argv[++i]);
        }
```

2-3 Debugging flag z

We first predefined a debugging flag u,c,z in debug.h

```
const char dbgSys = 'u';
const char dbgTraCode = 'c';
const char dbgExpr = 'z';
```

- [A] when insert into queue
  In Scheduler::PutIntoQueue(), we add a DEBUG() with flag dbgExpr.

```
Scheduler::PutIntoQueue(){
    DEBUG(dbgExpr, /
    "[InsertToQueue] Tick [" << kernel->stats->totalTicks
    << "]: Thread [" << newThread->getID() /
    << "] is inserted into queue L["<< layerIdx <<"]");
}
```

- [B] when remove from queue
  In Scheduler::RemoveFromQueue, we add a DEBUG() with flag dbgExpr.

```
Scheduler::RemoveFromQueue(){
    DEBUG(dbgExpr,
    "[RemoveFromQueue] Tick ["<< kernel->stats->totalTicks
    << "]: Thread [" << newThread->getID() /
    << "] is removed from queue L[" << layerIdx <<"]");
}
```

- [C] when change its scheduling priority
  In Scheduler::PerAgingProcess(), we add a DEBUG() with flagExpr.

```
Scheduler::PerAgingProcess(){
    DEBUG(dbgExpr,
    "[UpdatePriority] Tick ["<< kernel->stats->totalTicks
```

```
    << "]: Thread [" << iterThread->getID() /
    << "] changes its priority from ["<< foundPriority /
    << "] to ["<< iterThread->GetPriority() <<"]");
}
```

- [D] when thread updates its approximate burst time
  In Thread::TerminateBurstTimeCounting(), we add a DEBUG() with flag dbgExpr.

```
Thread::TerminateBurstTimeCounting(){ DEBUG(dbgE
    xpr, "[UpdateRemainingBurstTime] / Tick ["<<
    kernel->stats->totalTicks /
    << "]: Thread [" << ID << "] /
    update approximate burst time, from: ["/
    << predictTime <<"], add ["<< burstTime
    << "], to ["<< newPredictTime <<"]");
}
```

- [E] when context switch occurs
  In Scheduler::Run, we call DEBUG() with dbgExpr.

```
Scheduler::PerAgingProcess(){
    DEBUG(dbgExpr,
    "[ContextSwitch] Tick ["<< kernel->stats->totalTicks /
    << "]: Thread ["<< nextThread->getID() /
    << "] is now selected for execution, thread ["/
    << oldThread->getID() <<"] is replaced, /
    and it has executed [" /
    << oldThread->GetExecTick() << "] ticks");

}
```