Chapter 6

Kernelized Architecture Implementation

In this chapter we discuss the implementation of our scheduling algorithms within the framework of a kernelized architecture [TS92]. We begin by discussing the architecture. This is followed by a detailed description of the conservative and aggressive scheduling algorithms. We finally end the chapter with the major theorems and proofs. Unlike the trusted subject architecture, the kernelized one requires no proof for confidentiality.

6.1 Architecture

As mentioned before, kernelized architectures in general have no trusted subjects. In our architecture, there exists a single-level level manager process at each level to coordinate the various concurrent computations running at the respective levels. Figure 6.1 illustrates the kernelized architecture.

While security comes for free in a kernelized architecture (since there are no trusted subjects), the price we have to pay is the additional complexity involved in implementing the various scheduling algorithms. In particular, the lack of a global multilevel coordinator such as the session manager in the kernelized architecture, necessitates that we implement the scheduling schemes in a distributed fashion.
6.2 Scheduling Algorithms

We now describe the algorithms to implement both the conservative and aggressive scheduling schemes under a kernelized architecture. We begin by describing some level manager data structures that are common to both schemes.

Level manager data structures:

- **current-wstamp**: the current timestamp given to objects written at the level of the level manager;
- **queue**: a queue of message managers waiting to be activated;
- **terminate-history**: a list of ordered pairs (forkstamp, wstamp);

In addition to the above data structures for every level, a message manager utilizes the following:
**Procedure terminate-kern-cons**(lmmsgmgr, wstamp, forkstamp)

{  
% Let tt be the message manager that just terminated at level lmmsgmgr  
% Let lm be the level manager at level lmmsgmgr  

% Update local current write stamp from tt  
lm.current-wstamp ← wstamp  

% Update local Terminate-history with the forkstamp and wstamp of tt  
Append-terminate-history(terminate-history, forkstamp, wstamp);

If queue is not empty  
then  
   dequeue(queue, mm);  
   start(mm);  
Else  
   Send a WAKE-UP message to all immediate higher level managers;  
End-If
}

end procedure terminate-kern-cons;

Figure 6.2: Level manager algorithm for terminate processing

local-stamp: a local table of timestamp entries with an entry at each level dominated by the message manager and identifying the versions at the level that will be used to process read-down requests;

forkstamp: forkstamp issued by the parent message manager;

### 6.2.1 Conservative Scheduling Algorithms

We now discuss the algorithms to implement the conservative scheduling scheme. Let us begin by looking at how fork requests are processed. When a computation is forked, a new message manager is created, and it is unconditionally queued by the local level manager, as shown in procedure fork-kern-cons in figure 6.3. The procedure also initializes the forkstamp entry and the phase 1 entries of the local-stamp table passed on by the ancestors (this is the astamps parameter for the procedure). When a level manager is notified of the termination of a message manager at its level, it first
Procedure fork-kern-cons(level-parent, level-create, forkstamp, astamps)
{
%Let level-create be the level of the local message manager
Create a new message manager mm at level level-create;

%Record the forkstamp passed on by the parent
mm.forkstamp ← forkstamp

%Begin phase 1 of acquiring local-stamp entries
For (every level l ≤ level-parent)
do
    initialize mm.local-stamp table entries from astamps;
End-For

%This is a priority queue maintained in forkstamp order
enqueue(queue, mm);
} end procedure fork-kern-cons;

Figure 6.3: Level manager algorithm for fork processing

Procedure wake-up-kern-cons
{
%Proceed if the necessary condition has been met
If a WAKE-UP message has been received from all lower levels
then
    If the queue is not empty
    then
dequeue(queue, mm);
    start-kern(mm);
    else
        Send a WAKE-UP message to all immediate higher levels;
    End-If
End-If
} end procedure wake-up-kern-cons;

Figure 6.4: Level manager algorithm for wake-up processing
Procedure start-kern(nn)
{
    %Let nn represent the message manager to be started
    %Let lm represent the level manager managing nn

    %Complete phase 2 of acquiring local-stamp entries
    For (every level l lower than the level of nn for which no timestamp has been obtained so far)
        do
            nn.local-stamp[l] ← mm.wstamp;
            where mm is the message manager entry in the terminate-history at level l
            with max{forkstamp: forkstamp < nn.forkstamp}
        End-For

    %Update the write stamp (wstamp) from the level manager
    nn.wstamp ← lm.current-wstamp + 1;

    %Begin execution of the message manager nn
    execute(nn);
} end procedure start-kern;

Figure 6.5: Level manager algorithm for start processing
updates the local terminate-history. The level manager then dequeues and starts the next computation at the head of its local queue; if the queue is found to be empty, a WAKE-UP message is sent to all immediate higher levels (see procedure terminate-kern-cons in figure 6.2). Let us now see how WAKE-UP messages are processed. When a level manager receives a WAKE-UP message from each of the immediate lower levels in the security lattice, it dequeues its local queue and starts the next message manager; if the queue is empty, the WAKE-UP message is simply forwarded to all the immediate higher levels in the lattice, as shown in the procedure terminate-kern-cons in figure 6.2.

As in the trusted subject implementation, a message manager's local-stamp vector is once again initialized in two phases. The first phase entries are obtained as before from the ancestors. The second phase utilizes the terminate-history data structures at all levels dominated by the level of the starting message manager. Recall that this history contains a list of terminated computations identified by their forkstamps and their associated wstamp values at termination time. At each level, the computation with the largest forkstamp that is still less than the forkstamp of the message manager to be started, is selected, and the associated timestamp is read into the corresponding local-stamp entry.

We conclude this subsection by giving proofs of correctness and termination for our conservative level-by-level scheduling algorithms.

Proof of correctness.

Theorem 6.1 The conservative (level-by-level) scheduling algorithms for the kernelized architecture maintain the invariant inv-conservative.

Proof:
While there are two message manager algorithms, namely **send** and quit and four level manager algorithms fork, **start**, **terminate** and **wake-up**, we focus only the latter two for the proof. The **terminate** and the **wake-up** algorithms invoke the **start** procedure whereby computations get activated (started). It suffices therefore to show that these algorithms maintain the invariant **inv-conservative**.

Consider the **terminate** procedure first. If we assume that the invariant holds as a pre-condition before the procedure was invoked, then it follows that there are no active or queued computations at level lmsgmgr or lower. Now if the start-kern(mm) statement is reached, the following pre-conditions are true: (a) there exists one or more queued computations at level lmsgmgr; (b) the computation mm, with the lowest forkstamp will be started. The start-kern(mm) statement further ensures the post-condition: (c) mm, being the computation with the smallest forkstamp is started, and there are no queued or active computations at lower levels. This maintains the invariant. On the other hand, if the start-kern(mm) statement is not reached the invariant obviously continues to be true.

Consider the **wake-up** procedure next. From the **terminate** procedure we see that a WAKE-UP message is sent to all immediate higher levels only if there are no active or queued computations at or below the level that sent the message. Hence, when a WAKE-UP message has been received at a level say lwake, from all lower levels, the following are true:

d. there are no queued or active computations at levels lower than lwake;

e. there are no active or terminated computations at level lwake.

The latter condition is true since a computation can be started only as a result of a previous terminate at the same level or due to the receipt of a WAKE-UP message
(and no terminate event would have occurred at lwake at this point). Thus when
the statement start-kern(mm) is executed, the following post-condition is true: (f)
mm is the first computation to be activated at level lwake and there exists no queued
or active computations at levels lwake or lower. This clearly maintains the desired
invariant. Once again, if the start-kern(mm) is not reached, the invariant continues
to hold. □

Having shown how our algorithms maintain the invariant inv-conservative,
we now argue how these algorithms preserve serial correctness by maintaining cor-
rectness constraints 1, 2, and 3. We state this below as a corollary.

**Corollary 6.1** The conservative (level-by-level) scheduling implementation under in-
viant inv-conservative maintains serial correctness.

**Proof:**
When a computation c is started at a level l, the invariant inv-conservative re-
quires all computations that are forked at level l with smaller forkstamps, to have
terminated. This maintains correctness constraint 1. The invariant also requires
that on the start of computation c, all computations at levels lower than l to have
terminated. This requirement clearly maintains correctness constraint 2 since the
constraint requires only computations with smaller forkstamps than c and at levels
l or lower to have terminated. In other words, as far as lower level computations
are concerned, the invariant inv-conservative is more restrictive than correctness
constraint 2, and clearly maintains and implies the latter. Correctness constraint
3 has to do with versions assigned to process read-down requests. The arguments
given for the accumulation of local-stamp entries in the trusted subject architecture
still apply. Thus the local-stamp table entries collected in the first phase at fork
time by c reflect versions identifying the states of objects written at the level of these
ancestors before each successive child in the ancestral path was forked. The second phase entries as before, identify latest versions written at lower levels for which there were no ancestors. In summary, all read down operations that are mapped to the versions identified by the local-stamp entries, will read the same object states as in a sequential execution, and thus maintain correctness constraint 3.□

Proof of termination

We formally state as a theorem, that a session will eventually terminate.

Theorem 6.2 Under the conservative scheduling scheme, all computations in a session will eventually terminate and thus guarantee the termination of a user session.

Proof:

By induction on the number of security levels, \( n \), at which computations are forked in a session.

Basis: Consider the basis with \( n = 0 \). Then the only level with active computations will not have any fork requests emanating from it. It follows from the second part of the proof of lemma 5.1 that the session is guaranteed to terminate.

Inductive Step: For the induction hypothesis assume that when \( n \) is equal to \( m \), all computations terminate at the \( m \) levels and a WAKE-UP is sent to all immediate higher levels. For the inductive step consider \( m + 1 \) levels where level \( l_{m+1} \) is a maximal element in the security lattice and dominates a subset of the \( m \) levels. Now by the induction hypothesis, all computations at the \( m \) levels would have terminated and hence a WAKE-UP message would have been received at level \( l_{m+1} \) from all immediate lower levels in \( m \). It now remains for us to show that a WAKE-UP is received at level \( l_{m+1} \) from all immediate lower levels dominated by \( l_{m+1} \) that never had active computations in the user session. These levels thus do not belong to
The argument to show this can be made from the following: (1) The induction hypothesis guarantees that the root computations which are at the lowest level, say $l_1$, in $m$, would have terminated and sent a WAKE-UP message to all immediate higher levels; (2) WAKE-UP messages are always forwarded across empty levels. Hence all levels which dominate $l_1$ and in turn are dominated by $l_{m+1}$ would have WAKE-UP messages forwarded through them. This guarantees that $l_{m+1}$ would receive these messages from all immediate lower levels, and when this happens the computation at the head of the queue which has the smallest forkstamp will be dequeued and started. The termination of this first computation at level $l_{m+1}$ is again guaranteed by lemma 5.1 and leads to the startup of the next one in the queue. Every terminate results in the next computation in the queue to be subsequently started in turn. The queue will thus be progressively emptied in finite steps and all computations at level $l_{m+1}$ would have then terminated. Thus the entire session will terminate. □

### 6.2.2 Aggressive Scheduling Algorithms

Having discussed a conservative scheduling scheme, we now turn our attention to an aggressive scheduling scheme. The implementation algorithms for the aggressive scheduling scheme are given in figures 6.6, 6.7, and 6.8 (the start algorithm is the same as in figure 6.5 for the conservative scheme). In addition to the data structures needed to implement the conservative scheme, the aggressive one requires that every level manager maintain a fork-history consisting of a list of ordered pairs (forkstamp, level). This helps a level manager keep track of the fork requests generated at its level. We now elaborate on these algorithms.

When a computation is forked (see the if statement in figure 6.6), we have to decide if it can be started immediately. A forked computation is started immediately if there exists no non-terminated computations at lower levels and with smaller
Procedure fork-kern-agg (level-parent, level-create, forkstamp, astamps)
{
% Let level-create be the level of the local level manager
Create a new message manager mm at level-create;

% Record the forkstamp passed on by the parent
mm.forkstamp ← forkstamp

% Begin phase 1 of acquiring local-stamp entries
For (every level l ≤ level of the parent of mm)
do
    initialize mm.local-stamp table entries from astamps;
End-For

% Check to see if a forked computation can be started immediately
If ∀ l ≤ level(mm), ∃ any computation c : (c.forkstamp < mm.forkstamp
    ∧ c ∉ terminate history at l)
    then,
    start-kern(mm);
else
    % This is a priority queue maintained in forkstamp order
    enqueue(mm);
end-if
}
end procedure fork-kern-agg;

Figure 6.6: Processing fork requests under aggressive scheduling

Procedure wake-up-kern-agg
{
dequeue(queue, mm);
start-kern(mm);
}
end procedure wake-up-kern-agg;

Figure 6.7: Processing wake-up requests under aggressive scheduling
Procedure term-kern-agg(lmsgmgr, wstamp, forkstamp)
{
% Let tt be the message manager that just terminated at level lmsgmgr
% Let lm be the level manager at level lmsgmgr

% Update local current write stamp from tt
lm.current-wstamp ← current-stamp

% Update terminate history
Append-terminate-history(terminate-history, forkstamp, wstamp)

% Check if a computation at level lmsgmgr can be started
If queue is not empty
then
% Let mm be the computation at the head of the queue
If ∀l < level(mm), ¬∃ c : (c.forkstamp < mm.forkstamp
∧ c ∉ terminate history at level l)
then
dequeue(queue, mm);
start(mm);
End-If
End-If

% Check if a computation at levels ≥ lmsgmgr can be started
For all levels l ≤ lmsgmgr
do
If ∃ c ∈ fork-history at l with (level(c) > lmsgmgr ∧
c.forkstamp > tt.forkstamp):
¬∃ any computation k with (level(k) ≤ level(c) ∧ k.forkstamp
< c.forkstamp ∧ k ∉ terminate history at level(k) ∧ k is not an
ancestor of c)
% We checked to see if c was not preceded by a lower-level active or queued
% non-parent computation in any of the fork-histories searched
then
Send a WAKE-UP message to the level manager of c at level ll;
End-If
End-For
}
end procedure term-kern-agg;

Figure 6.8: Processing terminate requests under aggressive scheduling
forkstamps. We can determine all the computations forked at lower levels by examining the fork histories at these levels. We can further determine which of these computations have terminated by examining the terminate histories at these lower levels. When processing terminate requests, a similar check is made upon the termination of a computation at a level to see if the next computation, if any, at the head of the queue for this level, can be started (see figure 6.8). We also check to see if computations queued at higher levels can be released. We examine the fork histories at lower levels for computations that have been forked from these lower levels but have larger forkstamps than the just terminated computation (see the for statement in figure 6.8). Such computations with larger forkstamps can be started so long as they are not preceded by lower level non-terminated computations to the left, in the computation tree. A WAKE-UP message is sent to the level managers at the levels for which computations can be stared. On receiving such a message, a level manager dequeues and starts the next computation at the head of its queue (see figure 6.7).

We now give proofs of correctness and termination for the aggressive scheme.

**Proof of correctness.**

**Theorem 6.3** The aggressive scheduling algorithms for the kernelized architecture maintain the invariant inv-aggressive.

**Proof:**

We start with the fork-kern-agg procedure in figure 6.6. We see that for the statement start-kern(mm) to be executed, the following pre-conditions are true:

a. there exists no non-ancestral queued or active computations at or below level(mm) and with a smaller forkstamp than mm;

b. mm is the only computation at level(mm).
After computation mm has been started the condition (a) above still holds and thus the invariant is maintained. A similar argument can be made for the start-kern(mm) statement in procedure term-kern-agg. When mm is dequeued, condition (a) above holds, and since mm has the smallest forkstamp in the queue, the invariant is maintained after the execution of start-kern(mm).

It now remains to show that the start-up of a computation due to the receipt of a WAKE-UP message at a level, will not violate the invariant. To see this, we observe that a WAKE-UP message is sent to a higher level (in the terminate-aggressive procedure) only if there exists a pending computation say, c at the higher level that was denied immediate execution at fork time. Further, c has to have the smallest forkstamp among others at its level and should not be preceded by active or queued computations at lower levels and with a smaller fork than itself. Thus on receiving a WAKE-UP message, a level meets all the necessary conditions to start a computation. The post-condition following the start-kern(mm) statement in procedure wake-up-aggressive thus maintains the invariant. 

We now state and show how the invariant inv-aggressive maintains serial correctness under our implementation.

Corollary 6.2 The aggressive scheduling implementation maintains serial correctness.

Proof:
We basically have to show how the correctness constraints 1, 2, and 3 are maintained. For a computation to be dequeued and successfully started, invariant inv-aggressive requires all earlier forked computations at level l or lower, to have terminated. But this is what is precisely required to maintain correctness constraints 1 and 2. The
argument for the maintenance correctness constraint 3 is independent of the scheduling algorithm used. Thus the earlier argument given for the conservative case still holds. □

Proof of termination

Theorem 6.4 Under the aggressive scheduling scheme, all computations in a session will eventually terminate and thus guarantee the termination of a user session.

Proof:
To argue proof of termination for the aggressive algorithm, we observe that if a computation is denied immediate execution this can only be at fork time. Again we assume that once started, a computation is guaranteed to terminate (by lemma 5.1). Our task is thus basically to show that every queued computation will eventually be started. Now if on fork, a computation $f$ is denied immediate execution, then there must be at least one active computation say $c$, with a smaller stamp than $f$ and at or below level($f$). Now the termination of $c$ is guaranteed by lemma 5.1. The termination of $c$ will cause at least one computation with a greater forkstamp than $c$ and a smaller forkstamp than $f$, or $f$ itself, to be started. Now if $f$ is not started, there can only be a finite number of computations that can potentially block $f$. Subsequent terminate events will progressively decrease the number of such computations with a smaller forkstamp than $f$. This will result in the eventual release of $f$ for execution. With a similar argument, we can show that every queued computation will eventually be released for execution and thus run to termination. Thus the entire session will eventually terminate, concluding the proof. □