System Architecture

Daniel Stutz
http://www.use-strict.net

Version from April 8, 2004

Contents

1 Processes, Tasks and Threads 4
1.1 Processes 4
1.1.1 Events causing process creation 4
1.1.2 Conditions which terminate processes 4
1.1.3 Potential attributes of PCBs 5
1.2 Tasks and Threads 5
1.2.1 Benefits of Threads 5
1.2.2 Types of Threads 5
1.2.3 Multi-threading Models 6
1.2.4 Threading Issues 6
1.2.5 Thread Implementations 6
1.2.6 TCB Attributes 7
1.3 Thread Switch 7
1.3.1 Cooperative Scheduling 7
1.3.2 Pros and Cons of User-Level Threads 7
1.3.3 Events triggering Thread Switching 8
1.3.4 Thread Switch Environment 8
1.3.4.1 Example: Simplified Thread Switch implementation 8
1.3.5 Pros and Cons of Kernel-Level Threads 9
1.3.6 Thread States 9
1.4 Concurrent Threads 10
1.4.1 Interrelationships with Concurrent Threads 10
1.5 Signaling and Synchronization 10
1.5.1 Synchronization with Busy Waiting 10
1.5.2 Synchronization with Signal/Wait 11
1.5.3 Semaphores 11
1.5.3.1 Semaphore semantic for signaling 12
1.6 Mutual Exclusion 12
1.6.1 Peterson's Solution 13
1.6.2 Bakery Algorithm for n Threads 13
1.6.3 Drawbacks of Software Solutions 13
1.6.4 Hardware Solutions: interrupt disabling 14
1.6.5 Solution with Test-And-Set (TAS) 14
1.6.6 Semaphores 14
1.6.6.1 Semaphore semantic for mutual-exclusion 14
1.6.6.2 Strong/Weak Counting Semaphores 14
1.6.7 Monitors 14
1.6.7.1 Condition Variables 15

2 Communication (IPC) or Message Passing **16**
2.1 Synchronization of communicating partners . . . 16
2.1.1 Synchronization of Sender . . . 16
2.1.2 Synchronization of Receiver . . . 16
2.1.3 Combinations of Senders and Receivers . . . 16

3 Deadlock and Starvation **17**
3.1 Necessary Conditions for a Deadlock . . . 17
3.2 Two Phase Locking . . . 17
3.3 Deadlock Avoidance . . . 18
3.3.1 Invariants and Constraints . . . 18
3.3.2 Deadlock Avoidance Algorithm . . . 18
3.3.2.1 Safe State . . . 19
3.3.2.1.1 Banker Algorithm . . . 19
3.4 Deadlock Detection . . . 20
3.5 Deadlock Recovery . . . 20

4 Scheduling **21**
4.1 Quantitative Performance Goals . . . 21
4.2 Qualitative System Goals . . . 21
4.3 Classification of CPU Scheduling . . . 21
4.3.1 Long-Term Scheduling . . . 21
4.3.2 Medium-Term Scheduling . . . 21
4.3.3 Short-Term or CPU Scheduling . . . 21
4.4 Characteristics of Scheduling Policies . . . 22
4.5 The CPU-I/O-Cycle . . . 22
4.6 Scheduling Policies . . . 22
4.6.1 First Come First Served (FCFS) . . . 22
4.6.2 Last Come First Served (LCFS) . . . 22
4.6.3 Round Robin . . . 22
4.6.4 Priority Scheduling . . . 23
4.6.5 Shortest Job Next . . . 23
4.6.5.1 Estimating the required CPU Burst . . . 23
4.6.6 Shortest Remaining Job Next . . . 24
4.6.7 Highest Response Ration Next . . . 24
4.6.8 Multilevel Feedback . . . 24
4.6.9 Fair Share Scheduling . . . 24
4.7 Multiprocessor Scheduling . . . 25
4.7.1 Scheduling Interrupts . . . 25
4.7.2 Scheduling Threads and Tasks . . . 25

5 Memory Management **27**
5.1 Fixed Partitions . . . 27
5.2 Dynamic Partitions . . . 27
5.3 Types of Fragmentation . . . 27
5.3.1 External Fragmentation . . . 27
5.3.2 Internal Fragmentation . . . 27
5.4 Placement Algorithms for Dynamic Partitions . . . 27
5.4.1 First-Fit . . . 27
5.4.2 Next-Fit . . . 28
5.4.3 Best-Fit . . . 28
5.4.4 Worst-Fit . . . 28
5.5 Address Space . . . 28
5.6 Memory Management Design Parameters . . . 28

5.6.1 Examples of Memory Management . . . 29
5.6.1.1 Boundary Tag System . . . 29
5.6.1.2 Buddy System . . . 29
5.7 Virtual Memory . . . 30
5.7.1 Notions of Virtual Memory . . . 30
5.7.2 Paging . . . 30
5.7.3 Mapping . . . 30
5.7.4 The Page Table . . . 30
5.7.4.1 Page Table Structure . . . 31
5.7.4.1.1 Control Bits of a Page Table Entry (PTE) . . . 31
5.7.4.2 Multilevel Page Tables . . . 31
5.7.4.3 Inverted Page Table . . . 31
5.7.4.4 Translation Look Aside Buffer . . . 31
5.7.5 Performance Analysis of the Paging System . . . 32
5.7.6 The Page Size Issue . . . 32
5.7.7 Policies around Paging . . . 32
5.7.7.1 Fetch Policy . . . 32
5.7.7.2 Placement Policy . . . 32
5.7.7.3 Replacement Policy . . . 33
5.7.7.4 Cleaning Policy . . . 33
5.7.8 Replacement Algorithms . . . 33
5.7.8.1 FIFO Policy . . . 33
5.7.8.2 LRU Policy . . . 33
5.7.8.3 Clock Policy . . . 33
5.7.9 Page Frame Buffering . . . 34
5.7.10 Further Virtual Memory Design Criteria . . . 34
5.7.10.1 Resident Set Size . . . 34
5.7.10.2 Replacement Scope . . . 34
5.7.10.3 Load Control . . . 34
5.7.10.3.1 Task Suspension . . . 34
5.7.10.3.2 The Working Set Policy . . . 35

6 Resource Allocation Protocols **36**
6.1 Non Preemptive Critical Sections (NPCS) . . . 36
6.2 Priority Inheritance (PI) . . . 36
6.3 Priority-Ceiling Protocol(PCP) . . . 37
6.4 Stacked Priority-Ceiling Protocol (SPCP) . . . 37

7 I/O **38**
7.1 Disk Performance Parameters . . . 38
7.2 Disk Arm Scheduling Policies . . . 38
7.2.1 First Come First Served . . . 38
7.2.2 SCAN (Elevator) . . . 38
7.2.3 Circular-SCAN (CSCAN) . . . 38
7.2.4 N-step-SCAN . . . 38
7.2.5 FSCAN . . . 39
7.2.6 Shortest Seek Time First . . . 39
7.2.7 Shortest Service Time First . . . 39

Chapter 1

Processes, Tasks and Threads

1.1 Processes

1.1.1 Events causing process creation

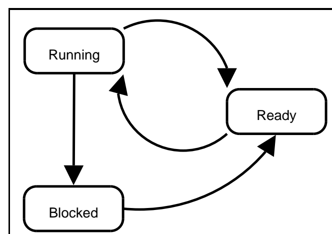
- System initialization
- System call within another process
- User command to create a new process
- Initiation of a batch job

1.1.2 Conditions which terminate processes

- Normal exit *(voluntary)*
- Error exit *(voluntary)*
- Fatal error *(involuntary)*
- Killed by another process *(involuntary)*

Process states

	Running	**Blocked**	**Ready**
Process blocks for input	Blocked	–	–
Scheduler picks another process	Ready	–	–
Scheduler picks this process	–	–	Running
Input becomes available	–	Ready	–



1.1.3 Potential attributes of PCBs

Process Management	Memory Management	File Management
Registers	Pointer to text segment	Root directory
Program Counter	Pointer to code segment	Working directory
Program Status Word	Pointer to stack segment	File descriptors
Stack Pointer		User ID
Process State		Group ID
Priority		
Scheduling Parameters		
Process ID		
Parent Process		
Process Group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

1.2 Tasks and Threads

Items shared by all threads in a task	Items private to each thread
Address space	Instruction pointer ("program counter")
Global variables	Registers, flags, etc. → context of thread
Open files	State
Child processes	
Pending alarms	
Signals and signal handlers	
Accounting information	

1.2.1 Benefits of Threads

- Responsiveness
- Resource sharing
- Economy
- Utilization of MP architectures

1.2.2 Types of Threads

- Kernel Level Threads
  - Known to the system wide thread management, often implemented inside the kernel,
    i.e. the corresponding TCBs are located inside the kernel.
- User Level Threads
  - Known only within one task or one subsystem, often implemented in a thread library,
    i.e. the corresponding UTCBs are locate inside an instance of the thread library.

1.2.3 Multi-threading Models

Many-to-One

Many user-level threads mapped to a single kernel-level thread. Used on systems, that
do not support kernel-level threads.

One-to-One

Each user-level thread maps to a separate kernel level thread.
(Windows, OS/2)

Many-to-Many

Allows many user-level threads to be mapped to many kernel-level threads.
Allows the OS to create sufficient number of kernel threads.

1.2.4 Threading Issues

- Semantics of fork() and exec() system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data

1.2.5 Thread Implementations

Windows 2000 Threads

Implements the one-to-one mapping. Each thread contains:

- a Thread ID
- register set
- separate user and kernel stacks
- private data storage area

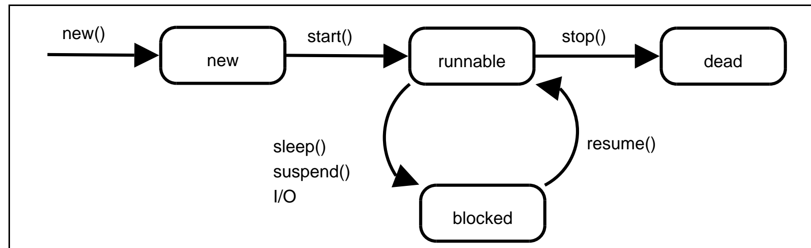
Linux Threads

Linux refers to them as tasks rather then threads. Thread creation is done
through the clone() system call. clone() allows a child task to share the
address space of the parent task (process).

Java Threads

Java threads may be create by either extending the Thread class or
implementing the Runnable interface. Java threads are managed by
the JVM.

Java Thread States

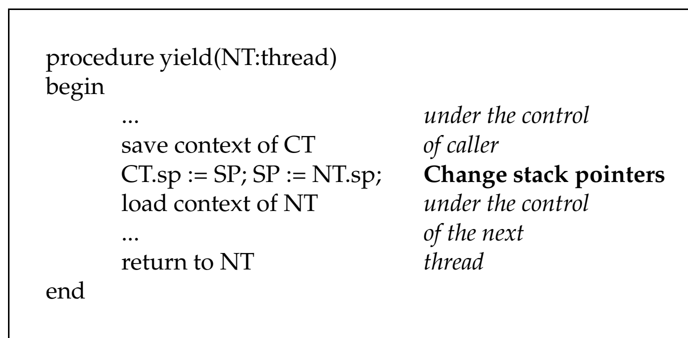


1.2.6 TCB Attributes

- minimal attributes
  - Thread identifier (TID)
  - Stack pointer (SP)
  - Status flags (SF)
- additional attributes
  - Context
  - Scheduling
  - Stack
  - additional private "resources"

1.3 Thread Switch

1.3.1 Cooperative Scheduling



```
procedure yield(NT:thread)
begin
        ...                              under the control
        save context of CT               of caller
        CT.sp := SP; SP := NT.sp;        Change stack pointers
        load context of NT               under the control
        ...                              of the next
        return to NT                     thread
end
```

1.3.2 Pros and Cons of User-Level Threads

Advantages	**Inconveniences**
Thread switch does not invoke the kernel ⇒ **no mode switching**	Many system calls are blocking ⇒ **all** threads of the task will be blocked
Scheduling policy can be application specific ⇒ **best fitting policy**	Kernel can only assign tasks to processors ⇒ 2 pure ULTs of the same task can never run on two processors simultaneously
ULTs can run on **any OS** if there is a **thread library**	

Inconveniences with Yield

- Cooperative scheduling requires *cooperative users*.
- Even with very cooperative users proper placing of the yield-calls is at least quite *cumbersome*
- Depending on the complexity of hardware and/or surrounding software there are a *lot of events* which have to be handled in between, anyway.

1.3.3 Events triggering Thread Switching

synchronous

- CT terminates
- CT calls synchronous I/O, must wait for the result
- CT waits for a message from another thread
- CT is cooperative, hands over CPU to another thread

asynchronous

- CT exceeds its time slice
- CT has lower priority, than another active thread
  - CT interrupted by a device waking up another thread
  - A higher priority thread's *sleep time* is exhausted
  - CT creates a new thread with higher priority
- exceptions
- interrupts

1.3.4 Thread Switch Environment

- *Kernel Entry* + *Mode Switch (User → Kernel)*
- Change state of old thread
- Select new thread (optional)
- *Thread context switch*
- Change state of new thread
- *Kernel Exit* + *Mode Switch (Kernel → User)*

1.3.4.1 Example: Simplified Thread Switch implementation

Assumption: HW automatically pushes SP, IP and flags of current thread (e.g. CT = T1) onto its *kernel stack* (implemented within its TCB, e.g. TCB1).

Algorithm:

- Current thread T1 is running
- Clock interrupt saving context of T1 onto its kernel stack and loading context of clock interrupt handler (CIH)
- CIH states end of time slice of T1
- CIH calling **thread_switch(T2)** (TSw)
- TSw saves kernel SP(T1) and loads kernel SP(T2)
- CIH loading context of T2
- Current thread T2 is running

Idle Thread To guarantee, that there exists always a runnable thread to switch to, an *idle thread* is introduced.

1.3.5 Pros and Cons of Kernel-Level Threads

Advantages	**Inconveniences**
Kernel can simultaneously schedule threads of same task on different processors.	Thread switching within one task involves the kernel $\Rightarrow$ 2 mode switches per thread switch!!
A blocking system call only blocks the calling thread, but no other thread from the same task.	This may result in a significant slow down!!
Even "kernel" tasks can be multi-threaded.	

1.3.6 Thread States

Potential Benefits

- It's faster to look after a thread with a certain *thread state* if you group all threads with same state into one subset.
  $\Rightarrow$ may improve *performance*
- In multiprocessor systems thread states are not only efficient, they are necessary!

Definition: "Thread State" expresses the relation of a thread towards other threads and/or towards system:

Running thread is *executing* on a processor

Ready thread is *runnable* (waiting for a processor)

Blocked thread is *waiting* for an event (i.e. end of I/O)

Further Useful States:

New OS has performed the necessary actions to create the thread

Exit termination moves thread to this state. Tables and other information are temporarily preserved for auxiliary programs. Thread gets deleted, when this data is no longer needed.

Need for Swapping Up to now, all threads are mapped to main memory. Even in a virtual memory based system, if to many threads are hold in RAM, performance will decrease significantly (*thrashing phenomenon*).
In a thrashing situation, OS will swap out some tasks to disk.

Swapping related Thread States

Blocked Suspend blocked threads having been swapped out to disk

Ready Suspend ready threads having been swapped out to disk

1.4 Concurrent Threads

Potential Problems with Concurrency

Threads may *share resources* and or *access common data* (either shared memory or files). *Race conditions* may occur, if the result of an application *depends* on execution sequence of threads. Concurrent threads may even induce *conflicts*, e.g. competition around *exclusive resources*. And if access to shared data isn't controlled, threads may produce *inconsistent data*.

1.4.1 Interrelationships with Concurrent Threads

Degree of Awareness	Relationship	Mutual Influence	Control Problems
Unaware of each other	Competition	Results are independent Timing may be affected	Deadlock Starvation
Indirectly aware of each other	Cooperation by Sharing	Results may be dependent Timing may be affected	Mutual Exclusions Deadlock Starvation Data Coherence
Directly aware of each other	Cooperation by Communication	Results may be dependent Timing may be affected	Deadlock Starvation Serialization

1.5 Signaling and Synchronization

1.5.1 Synchronization with Busy Waiting

Synchronizing with busy waiting is inefficient. It's up to the scheduler to end up inefficient busy waiting times. Busy waiting is only useful, if it is known that the waiting times are very short for whatever reason.
Use kernel API to avoid busy waiting:

- BLOCK()
- DEBLOCK()

- YIELD()

Example Implementation of BLOCK():

```
BLOCK(myself){
    block(CT,myself);          # state transition
    NT = Schedule();           # later chapter
    CT = Thread_Switch(NT);
    Assign(CT)                 # state transition
}
```

1.5.2 Synchronization with Signal/Wait

Pros	Cons
No extra mechanism for a related problem.	Complicated and very inefficient for n < 2 threads (*not scalable*) It's very likely to forget to synchronize with a new thread (*uncomfortable*) Low performance due to avoidable kernel calls (*not efficient*)

The operations signal() and wait should be *atomic* to avoid race conditions.

Unbuffered Signals

Each incoming signal may overwrite a previous one.
(e.g. flag or binary semaphore)

- **Pro:** Reaction on each signal
- **Con:** Deficient signaling source floods system

Buffered Signals

Each incoming signal is buffered until a potential waiting thread consumes this signal.

- **Pro:** Reaction only to the newest signal
- **Con:** Danger of lost signals

1.5.3 Semaphores

Definition: A *semaphore* S is a integer variable that, apart from initialization, can only be accessed by 2 *atomic* and *mutual-exclusive* operations:

P(S)	P $\sim$ pass
V(S)	V $\sim$ leave

To avoid *busy waiting*:
If thread cannot enter the *critical section* inside of *P(SP)*, then put in into a *blocked queue* waiting for an event. The occurrence of this event is signaled via *V(S)* by another thread.

1.5.3.1 Semaphore semantic for signaling

- A *positive* value of the counter indicates how many signals are pending
- A *negative* value of the counter indicates how many threads are waiting for a signal, i.e are queued within the semaphore object
- If *counter == 0*, no thread is waiting for a signal and no signal is pending

1.6 Mutual Exclusion

If a thread executes code manipulating shared data or accesses a exclusively usable resource, the thread is in a *critical section*. The Execution of a critical section (CS) must be *mutual exclusive*, i.e. at any time, only 1 thread is allowed to execute this critical section.
⇒ Each thread must request the permission to enter a critical section.
That means, a *serialization protocol* has to be established and offered, which guarantees, that the results of the involved threads are not dependent on the order, in which their execution phases have been interleaved.
4 requirements for a valid solution of a mutual exclusion problem:

1. No two processes simultaneously in critical region
   ⇒ *Exclusiveness*
2. No assumptions about speeds or numbers of CPUs
   ⇒ *Portability*
3. No process running outside its critical region may block another process
   ⇒ *Progress*
4. No process must wait forever to enter critical section
   ⇒ *Bounded Waiting*

If *all 3 main criteria* (mutual exclusion, progress, bounded waiting) are satisfied, then a valid solution will provide *robustness against failures* within the remainder section (RS) of a thread. (Failures within RS will not affect other threads.) However, *no valid solution* can ever provide robustness, if a thread fails within its critical section (CS), because *a thread failing within its CS may never perform exit_section(), i.e. no other thread related to that CS may ever perform enter_section() successfully!*

1.6.1 Peterson's Solution

Initialization:
flag[0] := flag[1] := false, and turn := 0;
willingness to enter *CS* specified by
flag[i] == true; If both threads attempt
to enter their *CS* simultaneously,
the turn value decides, which one will win.

```
thread Ti:
repeat
      flag[i] = true;
      turn = j;
      do() while
      (flag[j] and turn == j);
            CS
      flag[i] = false;
            RS
forever
```

1.6.2 Bakery Algorithm for n Threads

Before entering their CS, each T_i receives a *number*. The holder of the smallest number enters its CS. If T_i and T_j receive the same number and i<j, T_i is served first. T_i resets its number to 0 in its exit section.
Correctness relies on the following fact:
If T_i is in CS and T_k has already chosen its number[k] != 0,
then (number[i],i)<(number[k],k).

```
thread Ti:
repeat
      choosing[i] = true;
      number[i] = max(number[0],...,number[n-1]) + 1;
      choosing[i] = false;
      for( j = 0; j<n; j++ ){
            while( choosing[j] == true ) {};
            while( number[j] != 0 and (number[j],j)<(number[i],i) ) {};
      }
      CS
      number[i] = 0;
      RS
forever
```

1.6.3 Drawbacks of Software Solutions

- Threads requesting to enter a "locked critical section" are *busy waiting*.
- If critical sections have long execution phases it may be more efficient to block waiting threads.
- In a single processor system with static priority scheduling no solution with busy waiting works at all.

1.6.4 Hardware Solutions: interrupt disabling

Single Processor:
Mutual-exclusion is preserved but efficiency is degraded:
While in CS, we cannot interleave execution with other threads being in their RS.

- Delay of interrupt handling may affect hole system
- Application programmers may abuse this facility

Multi Processor:
Mutual-exclusion is not preserved.

1.6.5 Solution with Test-And-Set (TAS)

An algorithm that uses TAS for mutual-exclusion:

```
thread Ti:
repeat
    repeat {} until testandset(b);
    CS
    b = 0;
    RS
forever
```

1.6.6 Semaphores

1.6.6.1 Semaphore semantic for mutual-exclusion

- A *positive* value of the counter indicates that a thread may enter CS
- A *negative* value of the counter indicates how many threads are waiting to enter CS
- If *counter == 0*, one thread is in CS and no thread is waiting

1.6.6.2 Strong/Weak Counting Semaphores

Strong Counting Semaphores preserve "*First Come, First Served*", i.e. choose the *first* waiting thread in queue.
Weak Counting Semaphores choose *any* waiting thread from queue.

1.6.7 Monitors

A monitor is a high-level "language construct" with the semantic of a binary semaphore, but easier to control. It offers:

- one or more interface procedures
- a initialization sequence
- local data variables

Characteristics:

- local variables accessible only by monitor's interface procedures

- thread enters the monitor by invoking an interface procedure
- only one thread can be executed in the monitor at any time

A monitor already ensures *mutual-exclusion* ⇒ no need to program this constraint explicitly. Hence, *shared data* are *protected automatically* by placing them inside a monitor. Monitor locks its data whenever a thread enters.
Additional thread synchronization *inside the monitor* can be done by the programmer using *condition variables*. A condition variable represents a certain condition (e.g. an event) that has to be met before a thread may *continue* to execute one of the monitors procedures.

1.6.7.1 Condition Variables

Condition Variables are local to the monitor and can only be accessed by:

CondWait(cv) blocks execution of the calling thread on condition variable *cv*. This blocked thread can resume its execution only if another thread will execute **CondSignal(cv)**.

CondSignal(cv) resumes execution of some thread blocked on this condition variable *cv*. If there are several such threads: choose any one. If no such threads exists: do nothing.

Chapter 2

Communication (IPC) or Message Passing

2.1 Synchronization of communicating partners

2.1.1 Synchronization of Sender

- *Unsynchronized Send* (non-blocking)
  If no receiver waiting for message *forget* about message, *continue*
- *Asynchronous Send* (non-blocking)
  If no receiver waiting for message *deposit* message (if enough buffer place), *continue*
- *Synchronous Send* (blocking)
  If no receiver waiting for message, *deposit* message, *wait* for receiver

In all cases: If receiver is waiting for message, *transfer* message, *continue*

2.1.2 Synchronization of Receiver

- *Non-blocking* receive
  void if there is no message
- *Blocking* receive
  Waits, if no message available

2.1.3 Combinations of Senders and Receivers

	non-blocking receive	**blocking receive**
unsynchronized send	- bogus -	Sender polling
asynchronous send	Receiver polling	asynchronous communication
synchronous send	Receiver polling	Rendezvous

Chapter 3

Deadlock and Starvation

Formal Definition: A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

3.1 Necessary Conditions for a Deadlock

1. **Exclusiveness** No sharing at all.
   At least one resource must be held in a non-sharable mode. If another thread (task) requests that resource, it must be delayed until that resource is released.

2. **Hold and Wait** Allocating/releasing occur at random.
   There must exist a thread (task), that is holding at least one resource and that is waiting to acquire additional resources that are currently held by other threads (tasks).

3. **No Preemption** ...from exclusive resources.
   Resources cannot be preempted; a resource can be released only voluntary by the thread (task) currently holding it.

4. **Circular Wait** Circular dependency.
   There must exist a closed chain of threads (tasks) $\{T_1, T_2, ..., T_k\}$, such that each thread T_i holds at least one exclusive resource needed by T_i+1.

3.2 Two Phase Locking

- Phase One
  - precess tries to lock all records it needs, one at a time
  - if needed record found locked, start over
- Phase Two
  - performing updates
  - releasing locks

3.3 Deadlock Avoidance

At *runtime* a decision is made whether a complete new thread or a new request of an active thread might lead to a deadlock.
⇒ **2 straightforward policies:**

- Do not start a new thread if its requests all together with those of the current active threads *might* lead to a deadlock.
- Do not grant an incremented request to a thread if its allocation might lead to a deadlock.

Formal description of the system:
n threads T_1, T_2, ..., T_n and m resource types R_1, R_2, ..., R_m
$R = (r_1, r_2, ..., r_m)$ total amount of each resource
$V = (v_1, v_2, ..., v_m)$ total amount of each resource currently not allocated to one of the n threads.

$$C = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ . & . & \dots & . \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{pmatrix}$$ Maximal requirement per resource of each thread.

$$A = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ . & . & \dots & . \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{pmatrix}$$ Currently allocated resources of each thread.

3.3.1 Invariants and Constraints

$$R_i = V_i + \sum_{k=1}^{n} A_{ki}$$ for all i:
All resources are either available or allocated.

$C_{ki} \leq R_i,$ for all k, i — no thread claims more than the total amount of resources in the system

$A_{ki} \leq C_{ki},$ for all k, i — no thread has allocated more resources than initially claimed

3.3.2 Deadlock Avoidance Algorithm

Start a new thread if and only if:

$$R_i \geq C_{n+1,i} + \sum_{k=1}^{n} C_{ki} \text{ for all } k, i$$

This policy is *far to pessimistic* assuming that all threads will request *all their claimed resources at the same time*.
In practice, however, the following often holds:

- Some of the claimed resources are never requested
- Few threads need all their claimed resources at the same time

3.3.2.1 Safe State

The state of a system of n threads is safe if there is at least one execution sequence allowing that all n threads can complete.
Formal definition:
System state = **safe** $\Leftrightarrow$
$\exists$ permutation $< T_{k_1}, T_{k_2}, ..., T_{k_n} >$ within $\{T_1, T_2, ..., T_n\}$:

$$\text{for all } i \in \{1, 2, ..., n\} : C_{k_i} - A_{k_i} \leq V + \sum_{s=1}^{i-1} A_{k_s}$$

or

$$\text{for all } i \in \{1, 2, ..., n\} : C_{k_i} - A_{k_i} \leq R - \sum_{s=1}^{n} A_{k_s}$$

3.3.2.1.1 Banker Algorithm

```
type state = record
      R,V: array[0..m-1] of integer
      C,A: array[0..n-1,0..m-1] of integer
      T: {set of threads}
end

procedure deadlock_avoidance(var answer : state, DT : set of threads) {
      answer := undefined;   /* initialization */
      DT := T;               /* all threads deadlocked */
      while( answer == undefined ) {
            if ∃ T[i] ∈ DT: C[i] - A[i] ≤ V then {
                  DT := DT \ {T[i]};
                  V := V + A[i];
                  if DT ==  then answer := safe;
            }else{
                  answer := unsafe; /* n≥1 thread deadlocked */
            }
      }
}

O(n^2 m) with n = # of threads and m = # of resource types
```

3.4 Deadlock Detection

If neither prevention nor avoidance works, at least you want to detect deadlocks. The whole concept is a bit more optimistic.
Algorithm:
Replace in the Banker Algorithm the "if condition" as follows:

if $\exists$ *T[i]* $\in$ *DT: CR[i]* $\leq$ *V then* {

CR[0..n-1,0..m-1] is the matrix of the current *pending requests*. If there is a deadlock, DT = deadlocked threads

3.5 Deadlock Recovery

- **Recovery through preemption**
  - take a resource from another process
  - depends on nature of the resource
- **Recovery through rollback**
  - checkpoint a process periodically
  - use this saved state
  - restart the process if it is found deadlocked
- **Recovery through killing processes**
  - crudest but simplest way to break a deadlock
  - kill one of the processes in the deadlock cycle
  - the other processes get its resources
  - choose process that can be rerun form the beginning

Chapter 4

Scheduling

4.1 Quantitative Performance Goals

processor utilization	percentage a processor is not idle
throughput	number of threads completed per unit of time
turnaround time	time elapsed from the submission of a request to its completion
response time	time elapsed from the submission of a request to the beginning of the response
no deadline violation	meeting all deadlines if possible

4.2 Qualitative System Goals

predictability	low variance in turnaround times of a specific task
robustness	few system crashes
fairness	few starvation's

4.3 Classification of CPU Scheduling

4.3.1 Long-Term Scheduling

Determines which tasks are admitted to the system. Therefore, controls the degree of "multiprogramming". If more tasks are admitted it is less likely that all tasks will be blocked awaiting some event ⇒ better CPU usage. On the other hand each task has less fraction of the CPU ⇒ longer response time.
The *long term scheduler* may attempt to keep a mix of CPU-bound and I/O-bound tasks.

4.3.2 Medium-Term Scheduling

Makes *swapping decisions* based on the need to manage multiprogramming. Done by memory management software, respectively by some specialized regulating module.

4.3.3 Short-Term or CPU Scheduling

Determines which thread is going to be executed next. The short term scheduler is also known as dispatcher.

It is invoked on an event that may lead to choose another thread for execution:

- clock interrupts
- I/O interrupts
- system calls or traps
- signals

4.4 Characteristics of Scheduling Policies

The *selection function* determines which ready thread will run next. The *decision mode* specifies the events when the selection function may be executed:

- *Non-preemptive*
  Once a thread is running, it will continue until it terminates, or yields, or blocks.
- *Preemptive*
  A running thread may be preempted (put back to ready queue) when more urgent work has to be done. Allows better service since trivial threads no longer can monopolize CPU.

4.5 The CPU-I/O-Cycle

Threads require alternate usage of CPU and I/O in a repetitive fashion. Each cycle consists of a CPU burst followed by a (usually much longer) I/O burst. A thread either terminates voluntary during a CPU burst, but may be aborted during an I/O- or CPU-burst by another thread. CPU bound threads may have longer CPU bursts than I/O-bound threads.

4.6 Scheduling Policies

4.6.1 First Come First Served (FCFS)

scheduling function:	the thread that has been waiting the longest time in the ready queue
decision mode:	non-preemptive
drawbacks:	A thread not performing any I/O monopolizes the CPU. FCFS implicitly favor CPU bound threads.

4.6.2 Last Come First Served (LCFS)

scheduling function:	the thread that has been waiting the shortest time in the ready queue
decision mode:	non-preemptive

4.6.3 Round Robin

scheduling function:	same as FCFS
decision mode:	(“time”) preemptive A non cooperative thread is allowed to run until its time slice (TS) ends. *(TS $\in$ [1,100] ms)* Then a timer interrupt occurs, the running thread is put onto the ready queue again.
drawbacks:	Still favors CPU-bound threads. I/O-bound thread doesn‘t use up its TS.

"Haldar's solution": virtual round robin:

When a I/O has completed, the blocked thread is moved to an auxiliary queue which gets preference over the main ready queue. Such a thread being dispatched from the auxiliary queue, runs no longer than the basic time quantum "*minus*" the time it was running in its previous TS. It gets only the unused remainder of its last TS. TS has to be substantially larger tan the time required to handle the clock interrupt and perform the dispatching, but not to large (otherwise FCFS).

4.6.4 Priority Scheduling

scheduling function:	the ready thread with the highest priority
decision mode:	non-preemptive
drawbacks:	Danger of starvation and priority inversion.

4.6.5 Shortest Job Next

scheduling function:	the ready thread with shortest expected CPU burst time
decision mode:	non-preemptive
drawbacks:	Possibility of starvation for longer threads as long as there is a steady supply of shorter threads. Lack of preemption is not suited in a time sharing environment. Implicitly incorporates priorities: shorter jobs are given preferences.

4.6.5.1 Estimating the required CPU Burst

Let $T[i]$ be the execution time for the i-th instance of this thread, i.e. the actual duration of the i-th CPU burst of this thread.
Let $S[i]$ be the predicted value for the i-th CPU burst of this thread.
The simplest choice is:

$$S[n+1] = (1/n) \sum_{i=1..n} T[i]$$

To avoid recalculating the entire sum this can be rewritten as:

$$S[n+1] = (1/n)T[n] + ((n-1)/n)S[n]$$

This convex combination gives equal weight to each instance.
Recent instances are more likely to reflect future behavior. A common technique to reflect that is to use *exponential averaging*:

$$S[n+1] = \alpha T[n] + (1-\alpha)S[n]; \quad 0 < \alpha < 1$$

More weight is put on *recent instances* whenever $\alpha > 1/n$
Weights of *past instances* are decreasing exponentially:

$$S[n+1] = \alpha T[n] + (1-\alpha)\alpha T[n-1] + ... + (1-\alpha)^i \alpha T[n-i] + ... + (1-\alpha)^n S[1]$$

The predicted value of the first instance ($S[1]$) is not calculated, but usually set to 0 to give a standard priority to new threads.

4.6.6 Shortest Remaining Job Next

scheduling function:	the thread with shortest expected remaining CPU time
decision mode:	preemptive any new or deblocked thread with a shorter remaining CPU burst time will preempt the current running thread.
drawbacks:	Possibility of starvation for longer threads as long as there is a steady supply of shorter threads. Lack of preemption is not suited in a time sharing environment. Implicitly incorporates priorities: shorter jobs are given preferences.

4.6.7 Highest Response Ration Next

Response Ration:

$$r := (waiting_time + processing_time)/processing_time$$

scheduling function:	the ready thread with the highest response ration
decision mode:	non-preemptive
comment:	Shorter jobs are favored, however, longer jobs do not have to wait forever, because their response ratio increases the longer they wait.

4.6.8 Multilevel Feedback

Preemptive scheduling with "dynamic self adjusting priorities".
Several ready queue with decreasing priorities:
$P(RQ_0) > P(RQ_1) > ... > P(RQ_n)$
New threads are favored and therefore placed in RQ_0. When they need the full time quantum, they are placed in RQ_1.
When they again need the full time quantum they are placed in RQ_2 and so on, until they finally reach RQ_n.
Every time a thread having waited for the end of an I/O is deblocked due to the end of this I/O it is placed in the most favorable RQ_0.
$\Rightarrow$ I/O-bound threads will stay in higher priority queues. CPU-bound jobs will drift downwards.
Dispatcher chooses a thread for execution in RQ_i only if all higher priorized queues are empty. Hence extremely CPU-bound threads may starve.

4.6.9 Fair Share Scheduling

Processes (tasks) are divided into groups, group k gets a fraction W_k of the CPU-capacity. The priority $P_j[i]$ of the process j (belonging to group k) at time interval i is given by:

$$P_j[i] = B_j + (1/2)CPU_j[i-1] + GCPU_k[i-1]/(4W_k)$$

A high value means a low priority. The process with the highest priority is executed next.

$$
\begin{aligned}
B_j &= \text{base priority of process} \\
CPU_j[i] &= \text{Exponentially weighted average of processor usage} \\
&\quad \text{by process } j \text{ in time interval } i \\
GCPU_k[i] &= \text{Exponentially weighted average of processor usage} \\
&\quad \text{by group } k \text{ in time interval } i \\
CPU_j[i] &= (1/2)U_j[i-1] + (1/2)CPU_j[i-1] \\
GCPU_k[i] &= (1/2)GU_k[i-1] + (1/2)GCPU_k[i-1]
\end{aligned}
$$

where

$$
\begin{aligned}
U_j[i] &= \text{processor usage by process } j \text{ in interval } i \\
GU_j[i] &= \text{processor usage by group } k \text{ in interval } i
\end{aligned}
$$

4.7 Multiprocessor Scheduling

4.7.1 Scheduling Interrupts

- Interrupts can be handled on each processor. I/O-interrupts should be handled on the processor having initiated the previous I/O-activity, because the thread having initiated I/O may be bound to this processor or this thread still may have some cache footprints.
- Interrupts may be handled on a processor already handling a previous interrupt. This may save one mode switch form user/kernel but may postpone interrupt handling true to interrupt convoys.
- Interrupts may be handled on the processor with the lowest "priority activity" (i.e. the idle thread)

4.7.2 Scheduling Threads and Tasks

- *Single-threaded tasks*
  - scheduling single threaded tasks, sharing code or data to the same processor "may" reduce *cache loading time*
  - *anonymous scheduling* on any processor "may" reduce turnaround times
- *Mulit-threaded tasks*
  - scheduling all threads of a task *saves cache loading time*, but also reduces concurrent execution completely
  - scheduling threads of a task on as many CPUs as possible supports concurrency, but may lengthen cache loading time
  - scheduling threads of one task at the same time (*gang-scheduling*) may profit from parallel execution

Additional CPU-Scheduling Parameters:

1. Number of processors to be involved
   - 1 processor
   - $p' < p$ processors
   - all p processors

2. Precedence relation
3. Communication costs
   - Communication between threads on different processors has to be done via main memory
   - Communication between threads on the same processor could be done via caches

Chapter 5

Memory Management

5.1 Fixed Partitions

Poor usage of memory, because each task, no matter how small needs an entire partition ⇒ *internal fragmentation*. Suitable sized partitions lessen this problem, but internal fragmentation still holds.

5.2 Dynamic Partitions

Partitions are of variable length and number: Each task gets exactly as much memory as it requires. After a task terminates, "*memory holes*" may appear ⇒ *external fragmentation*. Must use *compaction* to shift tasks so their partitions are contiguous ⇒ all free memory is in on block.

5.3 Types of Fragmentation

5.3.1 External Fragmentation

Total memory space exists to satisfy a request, but it is not contiguous.

5.3.2 Internal Fragmentation

Allocated memory may be slightly larger than requested memory. This size difference is memory internal to a partition but it is not being used.

5.4 Placement Algorithms for Dynamic Partitions

5.4.1 First-Fit

Scan the list or bitmap for the first entry that fits. If it is larger in size, break it into allocated and free part.

- May have many processes load in the front end of memory, that must be searched over when trying to find a free block.
- May have lots of unusable holes at the beginning ⇒ *external fragmentation*

5.4.2 Next-Fit

Like First-Fit, except it begins its search form that point in the list or bitmap where the last request succeeded.

- More often allocates a block of memory at the end of memory where the largest block is found
- Largest block is broken up into smaller blocks
- Compaction is required to obtain a large block at the end of memory

5.4.3 Best-Fit

Choose that block, that is closest in size to the request.

- **Poor performance**
- Often has to search the complete list or bitmap
- Since smallest fitting block is chosen for a request, the smallest amount of fragmentation is left in the memory ⇒ compaction must be done more often

5.4.4 Worst-Fit

Choose that block, that is largest in size. The idea is to leave a usable new fragment over.

- **Poor performance**
- Often has to search the complete list or bitmap

5.5 Address Space

Logical Address Scope
The "logical address scope" limits the range of addresses a compiler, linker and loader may give to an executable. The maximal size of a logical address scope is limited by the *address width* of the CPU.
Logical Address Space (LAS)
A ("*logical*") *address space* is the range of address within a logical address scope accessible for an "executable task".
The main purpose of a logical address space is **protection**:
If a thread or task tries to reference a logical address not belonging to its logical address space an exception is raised to handle the *address violation*.
Logical Address Range
A *continuous portion* of a logical address space is a *logical address range*. (i.e. segments or pages)

Logical Address Range ⊂ Logical Address Space ⊂ Logical Address Scope

5.6 Memory Management Design Parameters

1. Sequence of allocate/release-operations
    - FIFO (*queue*)
    - LIFO (*stack*)

- arbitrary

2. Size of memory portions
   - identical size (*most buffering*)
   - fixed size (*reservoir of frequently used portions*)
   - exponential size (*buddy system*)
   - arbitrary

3. Management data structures
   - integrated within memory portion(s)
   - special memory portion(s)

4. Fragmentation
   - with(out) internal/external fragmentation

5. Allocating policy
   - First-Fit
   - Next-Fit
   - Best-Fit
   - Worst-Fit
   - Nearest-Fit

6. Reunification of released portions
   - immediate reunification with neighbored free blocks (if any)
   - lazy reunification

5.6.1 Examples of Memory Management

5.6.1.1 Boundary Tag System

- Operations in arbitrary order
- Arbitrary sized portions
- Integrated management data structures
- Allocation according to Best-Fit
- External fragmentation
- Immediate reunification

5.6.1.2 Buddy System

- Operations in arbitrary order
- Allocated portions of $2^0, 2^1, 2^2, 2^3, \ldots$
- Explicit management data structures
- Allocation according to Best-Fit
- Internal and External fragmentation
- Immediate (or lazy) reunification

5.7 Virtual Memory

Shortcomings of non-virtual memory:

1. Completely mapped address spaces lower the degree of multiprogramming
2. Address spaces which are larger than the virtual memory have to be handled by the application programmer (*overlay technique*).

Key idea: Instead of swapping complete address spaces, the operating system *automatically* maps those *pages* onto *page frames*, which are currently needed.
⇒ The application programmer is no longer involved in any overlaying.

5.7.1 Notions of Virtual Memory

Virtual Memory (VM)

- Not a physical device but an abstract concept
- Comprises all virtual address spaces

Virtual Address Space (VAS)

- Set of visible or defined virtual addresses
- Single Address Space systems use a single VAS for all tasks

Resident Set (RS)

- Set of pages of a task currently mapped to physical memory

Working Set (WS)

- Set of pages a task currently really needs.
  Those pages "should" be in main memory.

5.7.2 Paging

Page:	2^i sized fixed portion of virtual address space.
Page frame:	2^i sized portion of physical memory.

typically $i \in [10, 16]$

5.7.3 Mapping

Virtual Memory:

- Divided into equal sized pages
- Mapping is a translation between page and page frame
- Mappings are defined at runtime and can change

Physical Memory:

- Divided into equal sized page frames
- Some page frames may be reserved for special purpose

5.7.4 The Page Table

The page table contains page frame numbers for all mapped pages to support address translation. It may contain additional page-management information.

5.7.4.1 Page Table Structure

The page table is an fixed size array (one entry for each page) of page frame numbers. Each page table entry (PTE) may also have other bits to control paging. The page table of the current running task must be in main memory. The physical starting address of the page table of the currently running task is hold in a single register.

5.7.4.1.1 Control Bits of a Page Table Entry (PTE)

valid/present	indicating whether corresponding page is mapped or not. if it is in main memory, the PTE holds the *frame number* of this page in main memory. if not, the PTE may contain the *"background" address* of that page on *disk* (either *offset* in *executable file* or in *swap area*).
modified/dirty	indicates if page has been modified since the page has been swapped-in the last time. if no change has been made, the page does not have to be written on disk when it is replaced.
referenced	indicates if a page has been read or written
defined	indicates if a page belongs to a task/thread
cache enabled	indicates if a page frame should be cached
pinned	indicates if page is resident
protection	indicates allowed access modes
protection level	kernel or user page

5.7.4.2 Multilevel Page Tables

A linear page table requires n pages to be stored. To cut down space requirements of the needed page tables of an address space a tree structure may be used (in case of a sparse address space).
Drawbacks: The larger the address space, the more levels are needed, the slower is the address translation ⇒ further hardware support needed (TLBs).

5.7.4.3 Inverted Page Table

The size of an inverted page table (IPT) depends on the number of page frames which is fixed given the current main memory size. Task ID and virtual page number are used to obtain a hash table entry which points to a chain of IPT entries. A page fault occurs if no match is found.

5.7.4.4 Translation Look Aside Buffer

The translation look aside buffer is a associative cache for a multilevel page table. The (directory,page) part of a virtual address can be used to look up the frame number directly (if its in the cache) without having to access the directory- or 2nd-level page table. Given a virtual address the MMU examines the TLB. If the PTE is present in TLB, its frame number is retrieved and the physical address is formed. If the PTE is not found in the TLB the frame number is looked up in the page table and the TLB is updated.
Hardware controlled TLB:
On miss, hardware performs PT lookup and updates TLB (e.g. Pentium)
Software controlled TLB:
On miss, hardware raises TLB miss exception, exception handler updates TLB (e.g. MIPS)
Page Tables are per address space, i.e. either per task or per process ⇒ TLB has to

be flushed each time a thread of another task/process is scheduled. Because this causes a high context switching overhead modern architectures use *tagged TLBs*, where an address space ID (ASID) is added to the TLB entries.
TLB Effect:
Without TLB we have on average 2 physical memory references per virtual reference. With TLB (assuming a 99% hit ratio) the average number of physical references per virtual reference decreases: $0.99 * 1 + 0.01 * 2 = 1.01$

5.7.5 Performance Analysis of the Paging System

Given:
$$p_{pf} = \text{page fault probability}$$
$$at_m = \text{access time to main memory}$$
$$et_{pf} = \text{execution time for a page fault}$$

*What is the average access time (**eat**) to virtual memory?*
$$eat_{vm} = (1 - p_{pf}) * at_m + p_{pf} * et_{pf}$$

5.7.6 The Page Size Issue

Advantages of large pages:

- short page tables
- fewer TLB entries ⇒ increasing TLB hit ratio

Advantages of small pages:

- minimize internal fragmentation
- each page matches code/data that is actually used

Page faults decrease to 1, if the page size is equal to the size of the address space.
The page fault rate is also determined by the number of page frames allocated per task.

5.7.7 Policies around Paging

5.7.7.1 Fetch Policy

When to swap-in a page. 2 main policies:

- *Demand Paging*
  swaps in a page only when a reference to that page has raised a *page fault*.
- *Pre-Paging*
  swaps in more pages than just the demanded one.
  May improve disk I/O throughput by reading chunks.
  Pre-fetch only, if disk is idle.
  May *waste* I/O-bandwidth if a pre-fetched page will never be used.
  May destroy a current working set.
  Pre-paging is *speculative* with all *risks*

5.7.7.2 Placement Policy

Where to place a swapped-in page.
For *pure segmentation* systems: First-Fit, Next-Fit, etc. are possible choices. For *paging* systems: frame location is irrelevant since all frames are equivalent

5.7.7.3 Replacement Policy

Which page/page frame to use for a newly fetched page in case of memory shortage.
Optimal policy selects page frame out of a set of replaceable page frames for which time to next reference is maximal.
Impossible to implement.

- First In First Out (FIFO)
- Least Recently Used (LRU)
- Clock (second chance)
- Least Frequently Used (LFU)

5.7.7.4 Cleaning Policy

When to swap-out a *dirty* page.

- *Demand Cleaning*
  a page is swapped out only when it's frame has been selected for replacement, but then a task that has a page fault may have to wait for 2 page transfers (out and in)
- *Pre-Cleaning*
  modified pages are swapped to disk before their frames are needed, so that they can be swapped out in clusters, but it makes little sense to swap out many pages if the majority them will be modified again before they will be replaced
- *Page Buffering*
  pages chosen for replacement are maintained either in a free (unmodified) or in a modified list. Pages of the modified list can be swapped out *periodically* ⇒ not all dirty pages are swapped out, but only those chosen for next replacement. Swapping out is done in batch, may improve disk-I/O.

5.7.8 Replacement Algorithms

5.7.8.1 FIFO Policy

Treats page frames as a circular buffer. When the buffer is full, the *oldest page* is replaced.

5.7.8.2 LRU Policy

Replaces the page that has *not been referenced for the longest period of time*.
Expensive to implement

5.7.8.3 Clock Policy

Set of page frames to be replaced is considered as a circular buffer. Whenever a page frame is replaced, a pointer is set to point to the next page frame in buffer. Replace the first frame with unset referenced bit. During search for replacement, set referenced bit in referenced pages to 0 (page gets a second chance to be referenced again before the next replacement).

5.7.9 Page Frame Buffering

Pages to be replaced are kept in main memory for a while to guard against poorly performing replacement algorithms. Two lists are maintained for replacement:

- a *free page list* for frames having not been modified since the have brought in (no need to swap out)
- a *modified page list* for frames having been modified (need to be swapped out before replacing)

A new page selected for replacement is inserted into one list and is unmapped but remains in memory. (i.e. its valid bit is reset). If a page in one of these lists is referenced again, the page only has to be removed from the list and has to be remapped, but no expensive swap in from disk or swap out to disk is necessary. The list of modified pages permits to swap out modified pages in clusters.

5.7.10 Further Virtual Memory Design Criteria

5.7.10.1 Resident Set Size

The operating system has to decide how many page frames are allocated to a task (respectively to its address space). If too few frames are allocated the page fault rate is high. If to many frames are allocated the degree of multiprogramming is low.

- *Fixed-allocation policy*
  allocates a fixed number of frames remaining over runtime.
  Determined at load time depending on the type of application
- *Variable-allocation policy*
  The number of frames allocated to a task may vary over time. It may increase if page fault rate is to high or decrease if its very low. Requires more overhead to assess behaviour of an active task.

5.7.10.2 Replacement Scope

Local replacement policy:
Chooses only among those frames that are allocated to the task having issued the page fault. **Global replacement policy:**
Each unlocked frame in the whole system may be a candidate for replacement.

5.7.10.3 Load Control

If too few tasks are running, all tasks may be blocked and the processor will be idle. If to many tasks are running, the resident size of each task will be to small ⇒ page fault rate will increase ⇒ *trashing*
A working set or page fault frequency algorithm implicitly incorporates load control: only those tasks whose *resident sets* are sufficiently large are allowed to execute.

5.7.10.3.1 Task Suspension

Explicit load control sometimes requires that tasks are swapped out.
Selection criteria to deativate a task:

- *Faulting task*
  This task obiously does not have its working set in main memory, thus it will be blocked anyway.

- *Last task activated*
  This task is least likely to have its working set already resident.
- *Task with smallest resident set*
  This task requires the least future effort to reload.
- *Task with largest resident set*
  This task will deliver most free frames.

5.7.10.3.2 The Working Set Policy

The Working Set policy is a variable-allocation method with local scope based on the assumption of locality of references.
Notion: The *working set* for a task *at time* t, $W(\Delta, t)$, is the set of pages that have been referenced in the last Δ *virtual time units*.
The working set of a task first grows when it starts executing and then tends to stabilize due to the principle of locality. It may grow again when the task enters a new locality (transition period) up to a point where the working set contains pages from two localities. It then decreases again spending some time in the new locality.

Page Fault Frequency Model
Define an upper bound U and a lower bound L for page fault rates. Allocate more frame to a task if its fault rate is higher than U. Allocate less frames if its fault rate is lower then L. The resident set size should be close to the working set size W. A task is suspended if the page fault frequency is higher then U and no more free frames are available.

Chapter 6

Resource Allocation Protocols

6.1 Non Preemptive Critical Sections (NPCS)

Strategy:	Process holding a resource is not preemptable.
Deadlock:	prevented
Priority Inversion:	bound
Max. Blocking Time:	execution time of the longest critical section of all lower priority processes.

6.2 Priority Inheritance (PI)

Strategy:	When a high-priority process blocks, the blocking process inherits the *current* priority of the blocked process
Deadlock:	not prevented
Priority Inversion:	controlled
Max. Blocking Time:	duration of $min(n, m)$ critical sections, where n is the number of lower priority processes and m is the number of resources than can be used to block the process.

6.3 Priority-Ceiling Protocol(PCP)

The resources required by all processes are *known a priori*. The *priority ceiling of a resource* R_i is equal to the highest priority of all processes that use R_i. The *priority ceiling of the system* is the highest priority ceiling of all resources *currently in use*.

Strategy:	Priority inheritance applies as in PI. When a process J requests a resource R either: R is busy, J blocks or R is free $\Rightarrow$ If J's current priority is greater then the systems priority ceiling, J is allocated R. If J's current priority is lower or equal then the systems priority ceiling, J blocks, except if J already holds a resource whose ceiling is equal to the systems ceiling.
Deadlock:	prevented
Priority Inversion:	bound
Max. Blocking Time:	execution time of the longest lower priority conflicting critical section.

6.4 Stacked Priority-Ceiling Protocol (SPCP)

Strategy:	After a process is released, it is blocked from starting until its assigned priority is higher then the current system ceiling. Unblocked processes are preemptively priority scheduled according to their assigned priority. Whenever a process request a resource it receives it.
Deadlock:	prevented
Priority Inversion:	bounded
Max. Blocking Time:	slightly better then PCP.

Chapter 7

I/O

7.1 Disk Performance Parameters

To read from or write to a disk, the disk head must be positioned at the desired track and at the beginning of the desired sector.

- *Seek Time*
  Time it takes to position the head at the desired track.
- *Rotational Delay/Latency*
  Time it takes until the desired sector has been rotated to line up with the read/write head.
- *Access Time*
  Sum of seek time and rotational delay. The time it takes to get in position to read or write.

7.2 Disk Arm Scheduling Policies

7.2.1 First Come First Served

Manage disk requests as they come. Fair to all "disk clients" $\Rightarrow$ no starvation. Good for just a few concurrent processes/tasks with clustered requests. Performs "random scheduling" if there are many concurrent disk clients.

7.2.2 SCAN (Elevator)

Disk arm moves in one direction satisfying all pending requests until it reaches the last track in that direction, then direction of arm movement is reversed.

7.2.3 Circular-SCAN (CSCAN)

Like SCAN, but restricts scanning to one direction only. When last track has been visited, arm is moved at full speed to first track.

7.2.4 N-step-SCAN

Segments the disk request queue into sub-queues of lenght N. Sub-queues are processed one at a time, using SCAN. New requests are added to another queue.

7.2.5 FSCAN

Two queues, no limit on queue-length. One queue is empty for new requests.

7.2.6 Shortest Seek Time First

Select the disk I/O request that requires the least movement of the disk arm from its current position. Each request on the most neighbored track is serviced regardless of its potential delay due to rotational time.

7.2.7 Shortest Service Time First

Select disk I/O request that can be serviced with the minimal access time.

Index

Bakery Algorithm, 13
Banker Algorithm, 19
Best-Fit, 28
Boundary Tag System, 29
Buddy System, 29

compaction, 27
condition variables, 15
critical section, 12

deadlock, 17
- avoidance, 18
- avoidance algorithm, 18
- Banker Algorithm, 19
- definition, 17
- detection, 20
- necessary conditions, 17
- recovery, 20
- safe state, 19
- unsafe state, 19

disk arm scheduling, 38
- CSCAN, 38
- FCFS, 38
- FSCAN, 39
- N-step-SCAN, 38
- SCAN, 38
- shortest seek time first, 39
- shortest service time first, 39

disk performance parameters, 38

external fragmentation, 27

First-Fit, 27
fragmentation
- external, 27
- internal, 27

immediate reunification, 29
internal fragmentation, 27
inverted page table, 31

lazy reunification, 29

mapping, 30
memory management
- address space, 28
- boundary tag system, 29
- buddy system, 29
- compaction, 27
- design parameters, 28
- dynamic partitions, 27
- fixed partitions, 27
- logical address range, 28
- logical address scope, 28
- logical address space, 28
- placement algorithms, 27
  - Best-Fit, 28
  - First-Fit, 27
  - Next-Fit, 28
  - Worst-Fit, 28
- virtual memory, 30

monitor, 14
multi-threading, 6
multilevel page table, 31
mutual exclusion, 12
- hardware solutions, 14
  - interrupt disabling, 14
  - Test-And-Set, 14
- software solutions, 13
  - Bakery Algorithm, 13
  - Peterson's Algorithm, 13

Next-Fit, 28

P(S), 11
page size, 32
page table, 30
paging, 30
Peterson's Algorithm, 13
Process, 4
- attribute, 5
- initialization, 4
- PCB, 5
- states, 4
- termination, 4

receiver, 16
resident set, 30
resource allocation protocols, 36
- NPCS, 36, 37
- PCP, 37

PI, 36
reunification
immediate, 29
lazy, 29

safe state, 19
scheduling, 21
cooperative, 7
CPU scheduling, 21
CPU-I/O-Cycle, 22
fair share, 24
long-term, 21
medium-term, 21
multiprocessor, 25
interrupts, 25
threads and tasks, 25
performance goals, 21
policies, 22
characteristics, 22
policy
FCFS, 22
highest response ratio next, 24
LCFS, 22
multilevel feedback, 24
priority scheduling, 23
round robin, 22
shortest job next, 23
shortest remaining job next, 24
virtual round robin, 23
short-term, 21
system goals, 21
Semaphore
semantic, 14
strong, 14
weak, 14
semaphore, 11
sender, 16
signal(), 11
synchronization, 10, 16
receiver, 16
sender, 16

Task, 5
Thread, 5
attribute, 7
benefits, 5
concurrent, 10
interrelationship, 10
idle, 9
issues, 6
Java, 6
Linux, 6
multi-threading, 6
states, 9
switch, 7
TCB, 7
types, 5
kernel level, 5
user level, 5
Windows 2000, 6
Threads
signaling, 10
synchronization, 10
TLB, 31
translation look aside buffer, 31

unsafe state, 19

V(S), 11
virtual address space, 30
virtual memory, 30
further design criteria, 34
inverted page table, 31
load control, 34
mapping, 30
multilevel page table, 31
page fault frequency, 35
page frame buffering, 34
page size, 32
page table, 30
paging, 30
paging performance, 32
paging policies, 32
fetch policy, 32, 33
placement policy, 32
replacement policy, 33
replacement algorithms, 33
replacement scope, 34
translation look aside buffer, 31
working set, 35

wait(), 11
working set, 30
Worst-Fit, 28

yield(), 7