Operating Systems Concepts: Revision Lecture

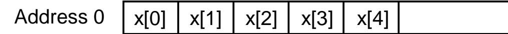
Olav Beckmann
Room 449 Huxley

Some General Comments

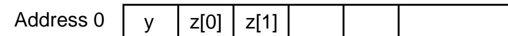
- **Make sure you have all the lecture handounts, especially make sure you look at tutorial exercises and notes on solutions – check web page**
- **Revision Guide – check web page**

Some Useful Facts

- Kilo, Mega, Giga:
  $1K = 2^{10} = 1024$
  $1M = 2^{20} = 1024 * 1024 = \ldots$
  $1G = 2^{30} = 1024 * 1024 * 1024 = \ldots$
- Storage layout of arrays in memory
  `int x[5];`

Address 0 

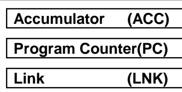
- Storage layout of structures
  typedef struct { int y, int z[2] } mydata;

Address 0 

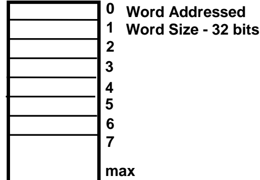
NARC - Not A Real Computer

Processor

Registers



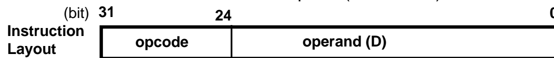
Memory



Word Addressed
Word Size - 32 bits

- Processor operates on data items - "words" stored in memory
- Each word consists of 32 binary (i.e. 0 or 1) digits - "bits"
- A word may represent a number (or characters or anything…)
- A word may represent an instruction for the processor
- Processor is a machine which interprets ("executes") instructions:

Instruction Layout



Maximum addressable memory ?

The NARC Instruction Set

Instruction opcode		*Meaning*
LOADC	1	ACC:= D
LOADM	2	ACC:= Memory(D)
STOREM	3	Memory(D):=ACC
ADDC	4	ACC:=ACC + D
ADDM	5	ACC:=ACC + Memory(D)
SUBC	6	ACC:=ACC - D
SUBM	7	ACC:=ACC - Memory(D)
JMP	8	PC:=D
JMPZ	9	If ACC=0 then PC:=D
JMPN	10	If ACC<0 then PC:=D
CALL	11	LNK:=PC; PC:=D
RET	12	PC:=LNK
HALT	13	

What exactly does the processor do?

```
int   Mem[MAX];  // main memory

// Internal registers of the processor
int   Acc;    // accumulator
int   Lnk;    // link register
int   PC;     // program counter
int   Op;     // current  opcode
int   D;      // current operand

void fetch() {
   int   W;
   W    = Mem[PC];
   Op   = Opcode(W);   // most significant 8 bits of W
   D    = Operand(W); // least significant 24 bits of W
}
```

This piece of C code __describes__ what the processor does to fetch an instruction

Execute

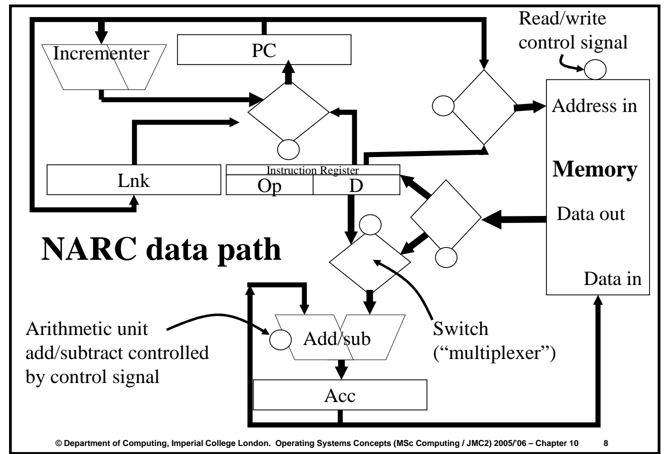
```
void execute( ) {
   switch(op) {                          // (opcode)
      case 1:    Acc = D;                // loadc
      case 2:    Acc = Mem[D];           // loadm
      case 3:    Mem(D) := Acc;          // storem
      case 4:    Acc = Acc+D;            // addc
      case 5:    Acc = Acc+Mem[D];       // addm
      case 6:    Acc = Acc-D;            // subc
      case 7:    Acc = Acc-Mem[D];       // subm
      case 8:    PC = D;                 // jmp
      case 9:    if (Acc=0) PC:=D;       // jmpz
      case 10:   if (Acc<0) PC:=D;       // jmpn
      case 11:   Lnk=PC; PC=D;           // call
      case 12:   PC=Lnk                  // ret
   }
}
```

This C function __describes__ what the processor does to execute an instruction

The "Fetch-Execute Cycle"

```
PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
} forever;
```

- **Execution starts at location zero (when NARC boots)**
- **Instructions are stored in memory**
- **So are the computation's working variables**
- **Fetch and execute are realised as digital circuits (see next slide)**



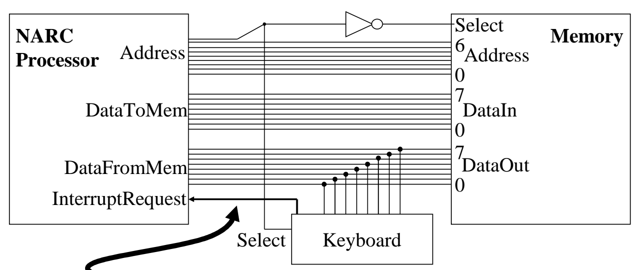
NARC - Control

- Each instruction in the NARC instruction set is implemented by a sequence of steps in which data is moved around the data path
- Eg. addm D:
  - **1:** AddressIn=PC; Read
  - **2:** Op&D = DataOut; PC=PC+1
  - **3** Switch(Op) {
  - **4:** AddressIn = D;
  - **5:** Acc=Acc+DataOut
- These steps are sometimes called "microinstructions"

An alternative to polling: **Interrupts**

- While we check the spelling of the preceding word, we want the keystrokes to be stashed in a buffer
- *"No matter how fast you type!"*
- There is a limit…
  - *incredibly brief keystrokes might not be noticed*
  - *two keystrokes in very very quick succession...*
  - *if the buffer is filled faster than it is emptied, it will fill up*
- I suppose we could build a special electronic box to fix it… or….

Interrupting the NARC processor



When a key is pressed, keyboard requests the processor to interrupt what it's doing and collect the keystroke

Modifying the fetch-execute cycle

```
PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
} forever;
```

```
PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
  if (InterruptRequest) {
    save(PC);
    PC = ipc;
  }
} forever;
```

Address of interrupt handler function

Not finished yet… why?

Enabling and disabling interrupts

```
PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
  if (InterruptRequest &&
      InterruptEnabled) {
    InterruptEnabled = false;
    save(PC);
    PC = ipc;
  }
} forever;
```

Block further interrupts; re-enable after dealing with this one

When do interrupts get re-enabled?

Saving the PC during an interrupt, and restoring it on return

```
PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
  if (InterruptRequest &&
      InterruptEnabled) {
    InterruptEnabled = false;
    Mem[0] = PC;
    PC = Mem[1];
  }
} forever;
```

Save the current PC somewhere (so we can return later), and branch to the address of the interrupt handler

The Interrupt Handler

```
PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
  if (InterruptRequest &&
      InterruptEnabled) {
    InterruptEnabled = false;
    Mem[0] = PC;
    PC = Mem[1];
  }
} forever;
```

```
void InterruptHandler() {
  saveProcessorState();
  char ch= *KBD_PORT_ADDRESS;
  KbdBuffer.add(scanToAscii(ch));
  restoreProcessorState();
  InterruptEnabled = true;
  PC=Mem[0];
}
```

Is this always going to work?

What if another interrupt occurs right away?

```
PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
  if (InterruptRequest &&
      InterruptEnabled) {
    InterruptEnabled = false;
    Mem[0] = PC;
    PC = Mem[1];
  }
} forever;
```

```
void InterruptHandler() {
  saveProcessorState();
  char ch= *KBD_PORT_ADDRESS;
  KbdBuffer.add(scanToAscii(ch));
  restoreProcessorState();
  rti; // restore PC from Mem[0] and
       // re-enable interrupts
}
```

We need a special instruction to do this indivisibly

Device driver structure

"Top half"

```
char getchar() {
  while (KbdBuffer.isEmpty())
    wait;   // maybe let another process run
  return KbdBuffer.remove();
}
```

"Bottom half"

```
void InterruptHandler() {
  saveProcessorState();
  switch (InterruptSource()) {
  case KBD:
       char ch= *KBD_PORT_ADDRESS;
       *KBD_PORT_ADDRESS=0; // acknowledge interrupt
       if (!KbdBuffer.isFull())
          KbdBuffer.add(scanToAscii(ch));
       break;
  default:  // code to handle other devices
  }
  restoreProcessorState();
  rti;
}
```


Device driver structure - top-half, bottom-half

- Top half of device driver is called by client process. It initiates device operation
- The two halves interact via a queue
- With an input device, the bottom half adds items to the queue, the top half removes them
- With an output device, the top half adds items to be output, the bottom half removes them

- ➢ Bottom half is initiated by interrupt signalling device completion
- ➢ Bottom half reinitiates device operation if there is more work to do
- ➢ **The top half can allow other processes to run while it is waiting**

Recall…

An interrupt can occur after the execution of any instruction

```
PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
  if (InterruptRequest &&
      InterruptEnabled) {
    InterruptEnabled = false;
    Mem[0] = PC;
    PC = Mem[1];
  }
} forever;
```

Saving the PC during an interrupt, and restoring it on return

Save the current PC somewhere (so we can return later), and branch to the address of the interrupt handler

Recall…

```
PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
  if (InterruptRequest &&
      InterruptEnabled) {
    InterruptEnabled = false;
    Mem[0] = PC;
    PC = Mem[1];
  }
} forever;
```

```
void InterruptHandler() {
  saveProcessorState();
  char ch= *KBD_PORT_ADDRESS;
  KbdBuffer.add(scanToAscii(ch));
  restoreProcessorState();
  rti; // restore PC from Mem[0] and
       // re-enable interrupts
}
```


Switching between processes

```
PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
  if (InterruptRequ
      InterruptEna
    InterruptEnab
    Mem[0] = PC;
    PC = Mem[1];
  }
} forever;
```

```
void InterruptHandler() {
  saveProcessorState();
```

1. Handle the interrupt
2. **Choose which processor state to return to**

```
  rti; // restore PC from Mem[0] and
       // re-enable interrupts
}
```


Critical Sections

- A critical section is a sequence of instructions which must be executed by at most one process at a time
- Analogy: a bridge strong enough for only one vehicle
- A code section is critical if it

1. **Reads a memory location which is shared with another process**
2. **Updates a shared memory location with a value which depends on what it read**

Semaphores

Process Interaction Mechanism #2

If processes are competing for some resource, a *semaphore* is a simple way of keeping track of the availability of that resource.

Binary Semaphore takes values: 0, 1
General Semaphore takes values: 0, 1, 2 …

The value of a Semaphore is a *protected* variable
– only accessible through two primitive operations:

```
P(s) ::= when s > 0 do s=s-1
V(s) ::= s=s+1
```

- **P means "I want" (Please)**
- **V means "Here is" (now 'Vailable)**

- **If value is 0 when you call P, P waits until some other process *- not you! -* calls V.**
- **The P & V operations are indivisible.**
- **As with locks, can be implemented using *busy-wait***
- **Also need initialization, init(s,v) ::= s:=v**

(***Proberen***, **to test,** ***verhogen***, **to increment)**

Non busy wait implementation of semaphores

Semaphore | **i**nteger | **Q**ueue | P1 | P2 | P3 | Waiting Processes

P(S)	**V(S)**

if S.i > 0 **then**
S.i := S.i-1
else
suspend process on S.Q
end if
	if not empty(S.Q) **then**
	resume a process in S.Q
	else
	S.i := S.i +1
	end if

- **The queue is usually First In First Out (FIFO).**

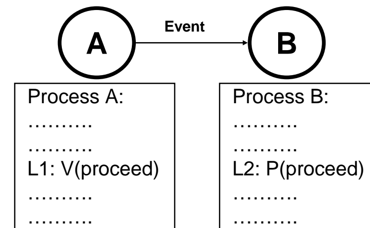
Using Semaphores:

Mutual Exclusion

```
var d: int          //shared variable
var s: semaphore
InitSema(s,1)    // initialise to 1

process p(n)
                  // s = 1
   P(s)           // s = 0
   d := d + 1
   V(s)
end p             // s = 1
```

Process can only enter critical section if s == 1.
Only one process at a time can be executing its critical section
- **so get *mutual exclusion.***

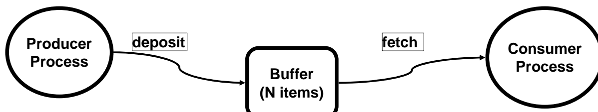
Using Semaphores: Synchronisation



- **Process B must wait at L2**
  - **until Process A reaches L1 and signals that B can proceed by executing *V(proceed).***
- **What value must the semaphore *proceed* be initialised to?**

Using Semaphores: Communication

Producer - Consumer problem – important example



***Three* semaphores for *three* "resources":**
- ***Space* in buffer is resource needed by Producer**
  - **allow deposit only when buffer not full (items in buffer < N)**
- ***Item* in buffer is resource needed by Consumer**
  - **allow fetch only when buffer not empty (items in buffer > 0)**
- **Mutual exclusion for buffer access is resource needed by everyone**
  - **allow buffer access only when no one else accessing it**

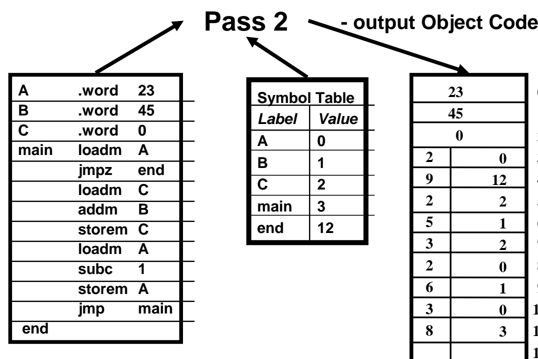
Semaphore Solution

```
var mutex: semaphore   // initialise to 1
var space: semaphore   // initialise to N
var item: semaphore    // initialise to 0

process Producer
  loop
     – produce item
     P(space)  // "I want space"
     P(mutex)  // "I want mutual exclusion"
     – deposit item
     V(mutex)  // "Here is mutual exclusion"
     V(item)   // "Here is item"
  end loop
end Producer

process Consumer
  loop
     P(item)    // "I want item"
     P(mutex)  // "I want mutual exclusion"
     – fetch item
     V(mutex)  // "Here is mutual exclusion"
     V(space)  // "Here is space"
     – consume item
  end loop
end Consumer
```

- **Solution still works for multiple Producer and Consumer processes.**
- **When *space* = 0 Producers cannot deposit items.**
- **When *item* = 0 Consumers cannot fetch items.**
- **What happens if we reverse the order of P operations in the Consumer?**

Two Pass Assembler

Pass 2 **- output Object Code**



A	.word	23
B	.word	45
C	.word	0
main	loadm	A
	jmpz	end
	loadm	C
	addm	B
	storem	C
	loadm	A
	subc	1
	storem	A
	jmp	main
end		

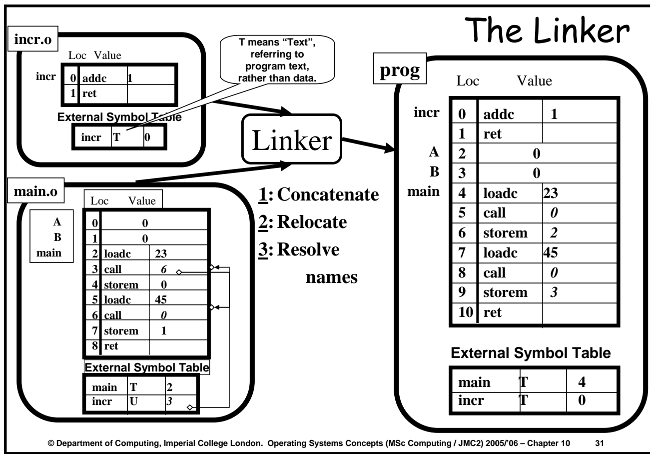
Symbol Table

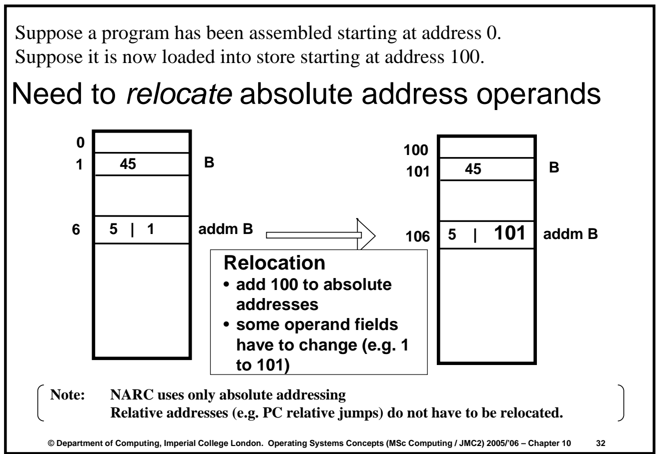
Label	*Value*
A	0
B	1
C	2
main	3
end	12

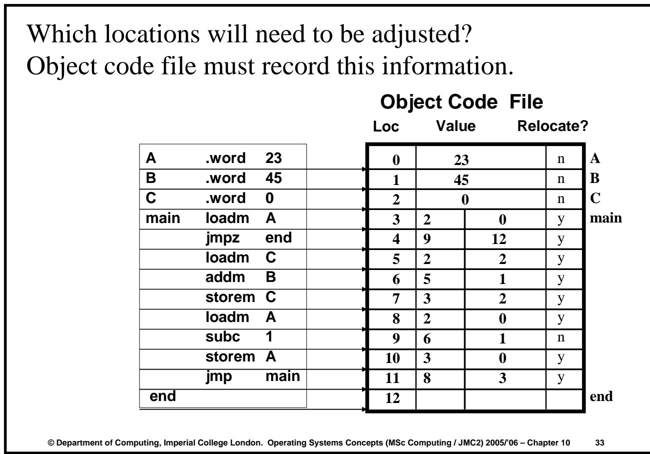
Address	Object Code	
0	23	
1	45	
2	0	
3	2	0
4	9	12
5	2	2
6	5	1
7	3	2
8	2	0
9	6	1
10	3	0
11	8	3
12		

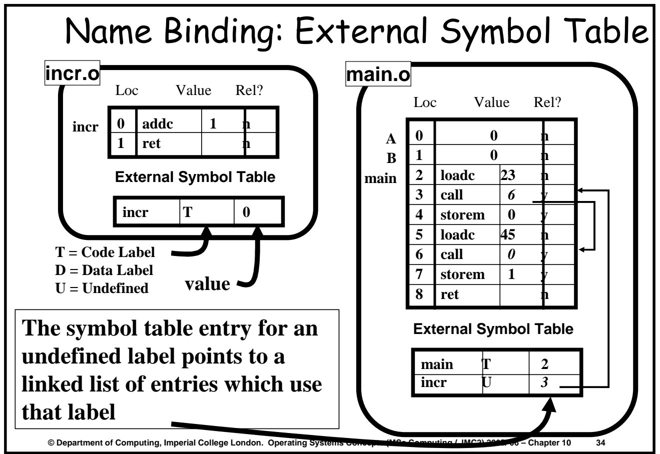
The Job of the Linker

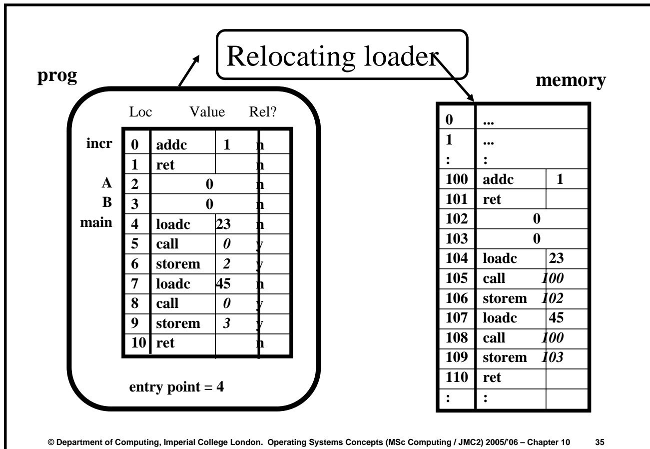
- Suppose we want our new program to use the functionality provided by another program
- E.g. newly-purchased word processor needs to access software driver for new keyboard
- Simpler working example: 'main' program uses separately-provided 'incr' procedure (next slide)
- Two issues:
  - **relocation**: concatenate the binary code - and adjust symbol references according to new addresses
  - **name binding**: resolve names used but undefined in 'main' with names defined in 'incr'

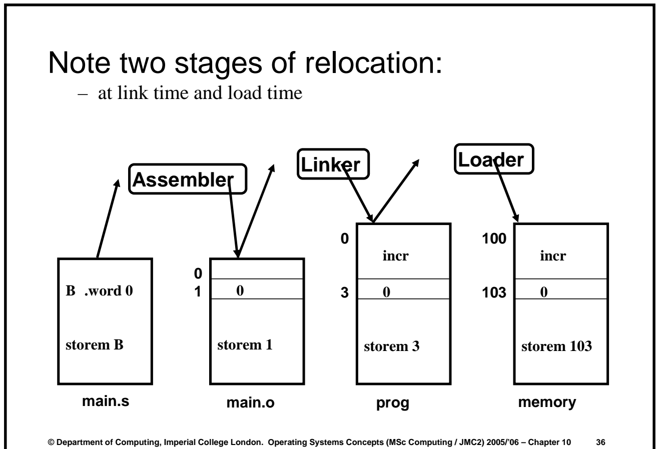

The Linker
incr.o
Loc Value
incr 0 addc 1
1 ret
External Symbol Table
incr T 0
T means "Text", referring to program text, rather than data.
Linker
main.o
Loc Value
A 0 0
B 1 0
main 2 loadc 23
3 call 6
4 storem 0
5 loadc 45
6 call 0
7 storem 1
8 ret
External Symbol Table
main T 2
incr U 3
1: Concatenate
2: Relocate
3: Resolve names
prog
Loc Value
incr 0 addc 1
1 ret
A 2 0
B 3 0
main 4 loadc 23
5 call 0
6 storem 2
7 loadc 45
8 call 0
9 storem 3
10 ret
External Symbol Table
main T 4
incr T 0
© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/'06 – Chapter 10 31


Suppose a program has been assembled starting at address 0.
Suppose it is now loaded into store starting at address 100.
Need to relocate absolute address operands
0
1 45 B
6 5 | 1 addm B
100
101 45 B
106 5 | 101 addm B
Relocation
• add 100 to absolute addresses
• some operand fields have to change (e.g. 1 to 101)
Note: NARC uses only absolute addressing
Relative addresses (e.g. PC relative jumps) do not have to be relocated.
© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/'06 – Chapter 10 32


Which locations will need to be adjusted?
Object code file must record this information.
Object Code File
Loc Value Relocate?
A .word 23
B .word 45
C .word 0
main loadm A
jmpz end
loadm C
addm B
storem C
loadm A
subc 1
storem A
jmp main
end
0 23 n A
1 45 n B
2 0 n C
3 2 0 y main
4 9 12 y
5 2 2 y
6 5 1 y
7 3 2 y
8 2 0 y
9 6 1 n
10 3 0 y
11 8 3 y
12 end
© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/'06 – Chapter 10 33


Name Binding: External Symbol Table
incr.o
Loc Value Rel?
incr 0 addc 1 n
1 ret n
External Symbol Table
incr T 0
T = Code Label
D = Data Label
U = Undefined
value
main.o
Loc Value Rel?
A 0 0 n
B 1 0 n
main 2 loadc 23 n
3 call 6 y
4 storem 0 y
5 loadc 45 n
6 call 0 y
7 storem 1 y
8 ret n
External Symbol Table
main T 2
incr U 3
The symbol table entry for an undefined label points to a linked list of entries which use that label
© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/'06 – Chapter 10 34

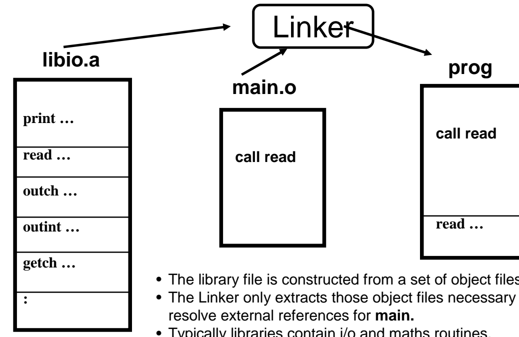

Relocating loader
prog
Loc Value Rel?
incr 0 addc 1 n
1 ret n
A 2 0 n
B 3 0 n
main 4 loadc 23 n
5 call 0 y
6 storem 2 y
7 loadc 45 n
8 call 0 y
9 storem 3 y
10 ret n
entry point = 4
memory
0 ...
1 ...
: :
100 addc 1
101 ret
102 0
103 0
104 loadc 23
105 call 100
106 storem 102
107 loadc 45
108 call 100
109 storem 103
110 ret
: :
© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/'06 – Chapter 10 35


Note two stages of relocation:
– at link time and load time
Assembler
Linker
Loader
B .word 0
storem B
main.s
0
1 0
storem 1
main.o
0 incr
3 0
storem 3
prog
100 incr
103 0
storem 103
memory
© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/'06 – Chapter 10 36

Linking, Loading and Relocation - Summary

- Assembler translates human-readable representation of instructions into object file:
  - object file includes list of external names defined, and names used but not defined
  - object file also includes relocation information
- Linker combines object files:
  - concatenate, relocate so internal name references OK
  - resolve name binding
- Loader finds sufficiently-large free memory region, interprets object file format, loads object file instructions and initialised data into memory
  - relocates so that internal name references are right

Library files - constructed by Archiver system utility



- The library file is constructed from a set of object files.
- The Linker only extracts those object files necessary to resolve external references for **main.**
- Typically libraries contain i/o and maths routines.

Kernel Interface

Today, kernels are usually written in a high-level programming language, with some low-level assembly code.

```
header "kernel.h" // not a real header
/* process synchronisation: */
    struct semaphore;
    void P(semaphore *s);
    void V(semaphore *s);
    initSema(semaphore *s, int value);

/* synchronization with real time: */
    int time();
    void delay(int ticks),

/* system set up: */
    int priority;
    typedef void Proc();
    void create(Proc *P, int size, int priority);
    void install(Proc *P, int intno);
    void startKernel();
```

create(P,size,pr) creates a new process executing function *P* with workspace *size* and priority *pr*.

time() returns clock ticks since kernel was started.

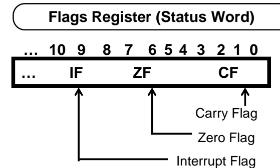
delay(t) suspends a process for *t* clock ticks.

install(P,intno) makes function *P* the handler for interrupt *intno*, so that *P* is invoked whenever that interrupt occurs.

startKernel() starts processes running under kernel control.

Enabling and Disabling Interrupts on Intel i386

Flags Register (Status Word)



Interrupt Flag: when set (equal to 1) interrupts are enabled otherwise they are disabled.

The only way to change the IF flag is through the following instructions:

cli	**clear IF (disable interrupts)**
sti	**set IF (enable interrupts)**
pushf	**put flags on stack**
popf	**set flags from top of stack**

- The eflags register carries "state" (information) between instructions
  - Zero flag – whether the result of a comparison instruction was zero; can be used by subsequent branch.
  - Carry flag – can be used to do "add with carry".
- The eflags register controls whether interrupts are enabled.
- For reasons of instruction set design, the eflags register cannot be directly addressed

Kernel Exit & Entry

To ensure mutual exclusion to the kernel data structures, kernel procedures must run with interrupts disabled. Only one process can be in the kernel at a time.

int int_mutexon()	disables interrupts, result is old value of eflags reg.
void int_mutexoff(int psw)	restores eflags register to value provided in psw

Assembler:

Assembler generated by gcc uses the accumulator register (or extended accumulator %eax) to return the result of some function calls. DON'T DO THIS AT HOME ☺.

```
.globl int_mutexon            .globl int_mutexoff
int_mutexon:                  int_mutexoff:
        pushf                         pushl 4(%esp)
        cli                           popf
        popl %eax                     ret
        ret
```

Kernel access procedures will have the following form:

```
void … (…){
  int psw = int_mutexon();      // disable  interrupts
                                     :
                                     :
  int_mutexoff(psw);            // restore interrupts
}                               //  to previous state
```

Why can't int_mutexoff simply enable interrupts?
Hint: what happens if one kernel procedure calls another?

Kernel Exit & Entry (cont.)

For convenience we will use the following macros:

```
#define ENTERKERNEL
                 int psw; psw = int_mutexon();

#define EXITKERNEL
                 int_mutexoff(psw);
```

See include/icos/intP.h in Simple Kernel source code.

The assembler routines int_mutexon and int_mutexoff (see previous slide) can be found in int_asm.S.

Semaphore Implementation

```
typedef struct Semaphore {
  int count;
  Queue waiting;
} Semaphore;
```

SEMAPHORE ← P, V, initSema

```
void P(Semaphore *s) {
  ENTERKERNEL
  if (s->count > 0)
    s->count--;
  else {
    addTail(s->waiting, running);
    dispatch();
  }
  EXITKERNEL
}
```

```
void V(Semaphore *s) {
  ENTERKERNEL
  if (isEmpty(s->waiting))
    s->count++;
  else {
    setReady(getHead(s->waiting));
    setReady(running);
    dispatch();
  }
  EXITKERNEL
}
```

```
void initSema(Semaphore *s, int value) {
  ENTERKERNEL
  s->count = value;
  initQ(s->waiting);
  EXITKERNEL
}
```


Key Datastructure in Kernel: FIFO Queue

QUEUE ← addTail, → getHead, → isEmpty, ← initQ

```
typedef struct Queue {
  Process head, tail;
} Queue;
```

Remove processes from head

struct Queue Q

head, tail → next → next → next

Add processes at tail of queue

Advantages of this data structure?

Ready Queue – Static multi-level priority

Actually *four* ready queues, one for each priority.

There must always be a process available to run.

Given a list of processes in the ready queues, which process will get run?

readyQ: High (3), Normal (2), Low (1), Idle (0) — Idle Process

```
Process running;    //points to currently running process
Queue readyQ[4];    //one queue for each priority level
```

```
enum Priority {high,normal,low,idle};
```


Scheduling Operations

Selecting a process to run

SCHEDULING (readyQ, running) ← setReady, dispatch, initSched

```
void setReady(Process p) {
  addTail(readyQ[int(p->prior)], p);
}

void dispatch() {
  Process oldrunning = running;
  int pr = 3;
  while (!isEmpty(readyQ[pr])) {
    pr--;
  }
  running = getHead(readyQ[pr]);
  psswitch( oldrunning->savearea,
            running->savearea);
}
```

```
void null() {
  while (1) {
        // idling loop---loop forever
        // Question: how does anything
        // else ever run?
  }
}

void initSched() {
  for (int pr=0; pr<4; pr++)
    InitQ(readyQ[int(pr)]);

  create(null, 0, idle);
  running = getHead(readyQ[int(idle)]);
}
```

Functionality of initSched is done by proc_init and startKernel in Simple Kernel

Process State Transitions

Delayed and *suspended* are both **blocked** states.

States: running, ready, delayed, suspended. Transitions: running → delayed: delay(t); running → suspended: P(s); running → ready: clock, V(s); ready → running: dispatch; delayed → ready: clock; suspended → ready: V(s); create → ready.

A process will be in exactly one of these states. The kernel maintains data structures to keep track of the processes in each state:

- A *ready* process – is on one of the four ready queues, readyQ[priority] (see below)
- A *delayed* process – is on delayQ
- A *suspended* process – is on the queue associated with a semaphore S
- The *running* process – is pointed to by the variable running (Just one processor, so only one process at a time in the running state.)

(4) Multi-Level Queues - Dynamic Priority Assignment

Unblocked processes → queue → CPU → Completion (100ms); pre-emption → queue → CPU → Completion (500ms); pre-emption → queue → CPU → Completion (2 secs); pre-emption

Increasing Priority ↑ / Increasing Time-slice Size ↓

- **Favours interactive jobs – they get a good response time.**

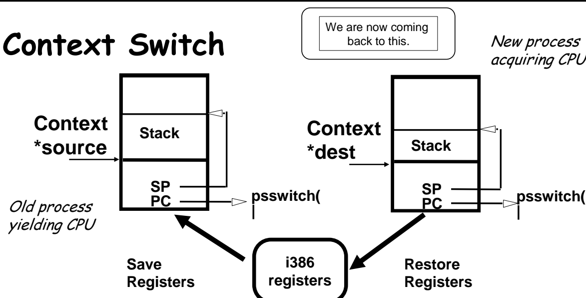
UNIX: Dynamic Priority Scheduling

- UNIX uses two-level scheduling (see lecture notes Ch 6).
- The UNIX low-level scheduling algorithm is a dynamic priority scheduling algorithm
- priority is an integer in the range [-20,20], with lower number meaning higher priority. Kernel mode processes negative, user mode processes positive.
- nice ("niceness") is also integer in range [-20,20]; added to priority. Users can choose positive niceness – lower priority, only superuser can choose negative niceness.
- Assume there are 40 priority queues; first process from highest queue gets to run.
- Quantum typically 100ms. After process uses up its quantum, rejoins its own priority queue -> round-robin for equal priorities.
- CPU_use_i = (quanta used in last second)*(CPU_usage / 2)

In real life: Linux Scheduling

- Linux scheduling is different from UNIX. Three classes:
  - "real-time" FIFO
  - "real-time" round-robin
  - Timesharing
- Timesharing: static priority value with default 20. Fixed relationship between priority and niceness: priority = 20 – nice.
- Users can alter (lower) priority by altering (raising) niceness
- quantum ("dynamic priority") starts at value of static priority, 20
  - quantum of running process is decremented by 1 at clock tick.
  - timer interrupt gives a clock tick every 10ms – "jiffy".
- goodness = quantum + priority if quantum > 0, else 0.
- When the scheduler is called, highest goodness is selected to run.
- Eventually, all threads (that are not blocked) will have

Context Switch

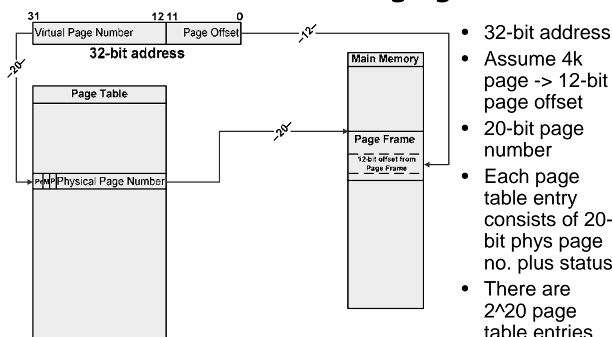



```
typedef struct Context { // Data structure for holding i386 registers
   int eax, ebx, ecx, edx, esi, edi, ebp, eflags, eip, esp;
   /*    0    4    8   12   16   20   24      28   32   36   note these offsets*/
} Context;

void psswitch(Context *source, Context *dest);
```


Memory Management

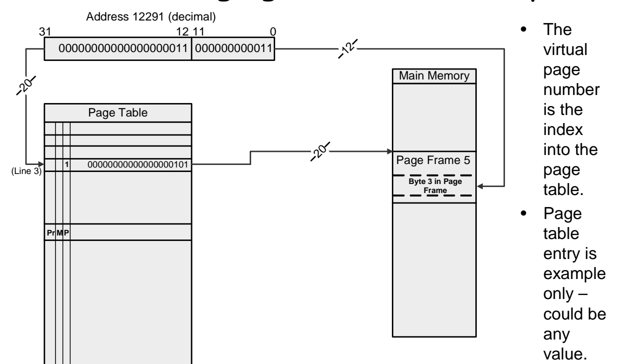
- Dynamic partition store allocation: each job is allocated a contiguous region of memory
  - Advantage over fixed partition: avoids wasted space
  - Disadvantage: fragmentation – several problems
  - Cannot run jobs larger than available memory
  - Dynamic partition store management with compaction means that we need virtual addressing
  - Placement algorithms (first fit, best fit etc) are about where to allocate a job, given a number of memory regions that are large enough

Basic Paging Scheme

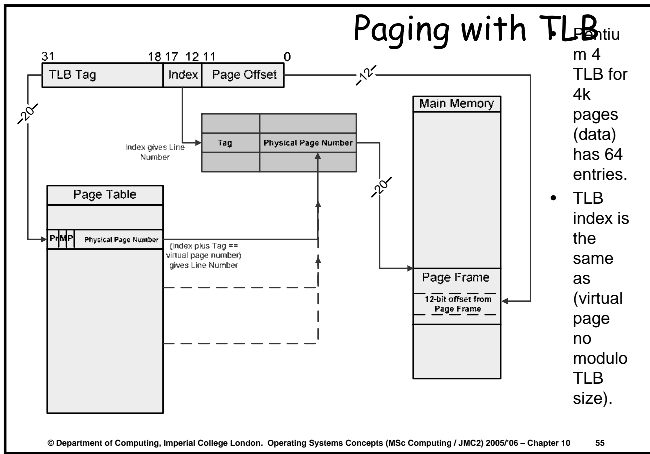


- 32-bit address
- Assume 4k page -> 12-bit page offset
- 20-bit page number
- Each page table entry consists of 20-bit phys page no. plus status
- There are 2^20 page table entries
- Page table size is at least (20 + status)

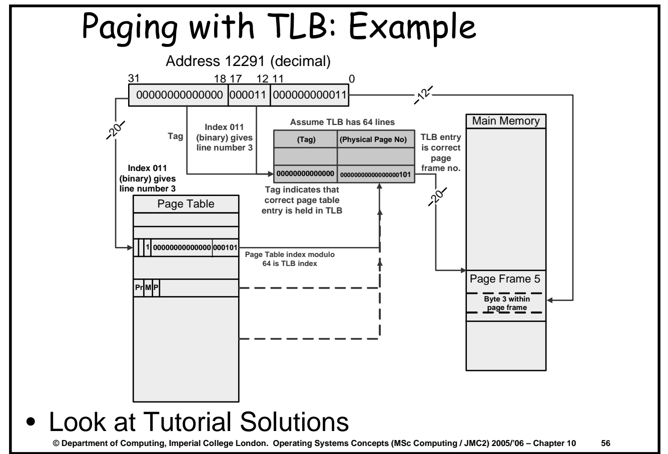
Basic Paging Scheme - Example



- The virtual page number is the index into the page table.
- Page table entry is example only – could be any value.



© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/'06 – Chapter 10



© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/'06 – Chapter 10

Summary

- Please work hard…
- When the exam starts –
  - Smile…
  - Think about the weighting of question parts
  - Manage your time!
  - It is much easier to get the first 30-40% of a question than to get the last 20% - therefore, make very sure you do not miss out on relatively easy-to-get marks by leaving too little time for one of the questions.
  - Take a look at the parts of the question you are answering – it should be obvious which part(s) are the easy parts.
  - Bring a watch, and allocate (at least initially) equal time to each question – they all carry equal weight!
- **When answering questions – think about the weighting: it is an indication of how much we expect you to say on a question.**
  - Questions are marked out of 20 (per question)
  - I.e. if a question-part counts 10%, that means 2 marks

© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/'06 – Chapter 10