Point of Sale Terminal

- standard POS terminal resembles:

```plaintext
  keypad  ┌───┐
        │    │
        v    v
  └───┘   ┌───┐
        │    │
        v    v
  └───┘   └───┘
        │    │
        v    v
  └───┘   └───┘
      printer  display
```

internal logic

bar code scanner
Point of Sale Terminal

- assignment 2 asks you to emulate a part of POS terminal:

  ![Diagram of POS terminal components]

Terminals

- a regular computer terminal:

  ![Diagram of computer terminal components]
Terminals: OS–eye view

- user types key
- bits arrive serially at UART in computer’s serial port
- when all bits received, generate interrupt
- kernel picks up byte of data from serial port and sends it...

UNIX Processes and ttys

- each process that is started at a terminal has associated with it a controlling tty
  - i.e., UNIX treats tty as a single device that is read/write capable

- each process has 3 files open for it:
  - stdin: where input is expected
  - stdout: where output is expected to go
  - stderr: where error output is expected to go
UNIX Processes and ttys

- Each process that is started at a terminal has associated with it a controlling tty.
- Each process has 3 files open for it.
  - Normally, a process begun from a controlling tty has:
    • stdin = keyboard
    • stdout, stderr = display

UNIX ttys

- Ttys normally operate in full duplex mode:
  • Consequence: received characters must be echoed back to display in order for user to see them.
  • \( \Rightarrow \) Tty driver program must perform echo operation.

- Other things this program does?
UNIX ttys

- line buffering: programs reading stdin only get complete line-at-a-time,
  - e.g., fread(mybuffer,1,256,stdin) returns only when a complete line (256 chars) is read
    - or EOF, or error

- ever use DEL or BS to correct a typing mistake?
  - tty driver understands these special characters and performs the ‘edit’ operation

- ever use ctrl-C to stop a program?
  - tty driver understands that character to mean send a SIGINT signal to the process

- ⇒ tty driver processes ‘special’ characters and takes action
UNIX ttys

- Some chars cause local edit operations
- Some chars cause specific actions to occur

Normal tty settings

- Terminal characteristics: (use "stty -a" command)
  - Speed 38400 baud, rows = 67, columns = 108, ...
- Special chars:
  - intr = ^c           quit = ^\           erase = ^h           kill = ^u
  - eof = ^d           eol = ^?           eol2 = ^?            swtch = ^
  - start = ^q          stop = ^s          susp = ^z          dsusp = ^y
  - rprnt = ^r          flush = ^o          werase = ^w         lnnext = ^v
- Modes:
  - parenb - parodd - cs8 - cstopb - hupcl - cread ...
  - -ignbrk - brkint - ignpar - parmrk - inpck - istrip ...
  - ixon - ixany - ixoff - imaxbel - isig - icanon ...
  - -xcase - echo - echoe - echok - tostop - echoctl ...
  - -echopr - echoke - defecho - flusho - opost - olcuc ...
  - onlcr - ocrnl - onocr - onlret - ofill - ofdel ...
Normal tty settings

- all aspects of tty operation are controllable
  - via appropriate system calls

UNIX devices

- all devices are represented as files
  - may be opened or closed
  - may be read and/or written
  - may be “controlled”
    - e.g., seek command on a disk drive

```c
int fd = open(filename,flags,mode);
```

- full path to file to be opened, e.g., dev/hda1
- read, write, r/w, etc.
- setting access modes
UNIX device control

- basic system call:
  - `int ioctl(int fd, int req, ...);

- valid requests depends on type of device (see, e.g., `man ioctl_list`)

- many categories of device have `ioctl()` wrappers
  - easier to work with

UNIX device control

- for ttys use `termios` functions (`man termios`)
- e.g.,
  - `int tcgetattr(int fd, struct termios *termios_p);
    - `int tcsetattr(int fd, int optional_actions, struct termios *termios_p);

- use for, e.g., control of echoing, character interpretation, ...
Deadlock

- 2 processes each waiting for the other to release a resource being held so processes can advance
  - generalizes to ≥ 2 processes, ≥ 2 resources

- problem occurs when processes follow a request → use → release model of resource utilization
  - i.e., exclusive use

Deadlock

- resources may be hardware
  - e.g., memory
  - assume 200 Kb of memory are available:

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>request 80 Kb</td>
<td>request 70 Kb</td>
</tr>
<tr>
<td>request 60 Kb</td>
<td>request 80 Kb</td>
</tr>
</tbody>
</table>

© Charles M Snow 9
Deadlock

- resources may be software
  - e.g., resources created/destroyed as needed, like signals, messages, data buffers

```
P1
...
receive(P2,M)
...
send(P2,N)
```
```
P2
...
receive(P1,Q)
...
send(P1,R)
```

Conditions for deadlock

1. mutual exclusion
   - only one process may use resource at a time
Conditions for deadlock

1. mutual exclusion
   - only one process may use resource at a time

2. hold and wait
   - a process may hold allocated resources while waiting to receive others requested

3. no pre-emption
   - no resource may be forcibly taken from a process holding it
Conditions for deadlock

- these 3 conditions are policy issues you design into your OS
- if you have all 3 of them, then deadlock occurs when:

4. circular wait
   - a closed chain of processes such that each holds
     \[ \geq 1 \] needed by next process in chain

![Diagram of processes and resources](image)

Depicting Deadlock

- resource allocation graphs
  - nodes:
    - type \( \star \) for processes
    - type \( R \) for resources
  - edges:
    - request \( P_i \rightarrow R_j \)
    - assignment \( R_j 

\rightarrow P_i \)
Depicting Deadlock: RAGs

- a resource class may have $\geq 1$ instance of that resource type
  - show as dots within rectangle for resource node

- absence of a cycle in a RAG $\Rightarrow$ no deadlock
- presence of a cycle $\Rightarrow$ is deadlock?
Depicting Deadlock: RAGs

• absence of a cycle in a RAG $\Rightarrow$ no deadlock
• presence of a cycle $\Rightarrow$ is deadlock?
  • if each R has only 1 instance, then yes
  • otherwise: may mean deadlock, may not

no cycle here: no deadlock
Depicting Deadlock: RAGs

- now cycle here: now deadlock

![Diagram showing a cycle leading to deadlock]

Depicting Deadlock: RAGs

- now cycle here: no deadlock

![Diagram showing a cycle leading to no deadlock]
Deadlock: What Can You Do?

- no really satisfying answers, but:
  1. prevention
  2. detection
  3. avoidance

Preventing Deadlock

- prevention: prevent any of the four conditions from occurring
  - indirect: prevent any of conditions 1, 2, or 3
  - direct: prevent condition 4
Preventing Deadlock

- disallow mutual exclusion
  - not really feasible
  - the nature of some resources precludes sharing them: must be used one-at-a-time

⇒ reject as solution

Preventing Deadlock

- hold & wait: prevent by
  - require a process to request all of its resources before it begins running
  - block the process until all its requests can be granted simultaneously

- or
  - a process requesting resources is not allowed to be holding any
  - if it has some and wants more, it must first release all held resources, then request
Preventing Deadlock

- effective
- wasteful:
  - a process may request a resource that it only uses late in execution, but holds throughout (no one else can use)
  - a process may wait ‘a long time’ for its resource request(s) to be granted (starvation)

⇒ OK, works, but not terrific

Preventing Deadlock

- no pre-emption: allow pre-emption
  - a process requesting resources may be denied and have resources taken away from it
  - it must re-acquire these later to continue
- or
  - a process already holding resources is denied further requests because another process is already waiting for [other] resources
  - we pre-emptively take resources away from waiting proc and give them to new requestor

⇒ works, but not really useful for many resources
Preventing Deadlock

- **circular wait:**
  - assign an ordering to resource types
  - require requests to be made in order

Does this work?
- let $i < j$, so $R_i$ precedes $R_j$
- let $P_A$ have $R_i$ and need $R_j$
- let $P_B$ have $R_j$ and need $R_i$
- hence have deadlock

$⇒ i < j$ and $i > j$

$∴$ works at preventing deadlock

---

Preventing Deadlock

- **circular wait:**
  - assign an ordering to resource types
  - require requests to be made in order
  - but wasteful, as was hold & wait solution
Detecting Deadlock

- idea:
  - let any process request and hold any resource any time
  - i.e., let deadlock occur if it’s going to
  - run some algorithm to detect such deadlock

- what kind of algorithm?

Detecting Deadlock

- what kind of algorithm?
  - algorithm able to find cycles in graphs could find deadlock
  - start with RAG:
    - delete R nodes
    - collapse assignment edges accordingly
    - result is wait-for-graph (WFG)
  - if there is a cycle in WFG, there is deadlock
RAGs to WFGs

Detecting Deadlock

- this works for single-instance only (SG&G 8.6)
- OS must maintain graph and run cycle-detection algorithm periodically
  - \( O(n^2) \): expensive
  - graph takes up resources
- multi-instance case: SG&G, page 251
Deadlock Detected: now what?

- so you detect deadlock, now what?
- several strategies possible:

Deadlock Detected, now what?

1. abort all deadlocked processes
   - most common strategy
Deadlock Detected, now what?

1. abort all deadlocked processes
2. roll–back each deadlocked process to some previous checkpoint, then re–run them
   1. need to have ability to roll–back and restart processes
   2. no guarantee previous dead–lock doesn’t re–occur
3. abort deadlocked processes one by one until deadlock goes away
   - use some “minimal–cost” metric to guide selection of processes
Deadlock Detected, now what?

1. abort all deadlocked processes
2. roll-back each deadlocked process to some previous checkpoint, then re-run them
3. abort deadlocked processes one by one until deadlock goes away
   - use some “minimal-cost” metric to guide selection of processes
     - least CPU used so far
     - lowest priority
     - highest estimated time remaining
   - after each proc is killed, need to re-run detection algorithm to test if done
Deadlock Detected, now what?

1. abort all deadlocked processes
2. roll-back each deadlocked process to some previous checkpoint, then re-run them
3. abort deadlocked processes one by one until deadlock goes away
4. pre-empt resources until deadlock goes away
   - roll process back to point before it acquired resource

Deadlock Avoidance

- prevention
  - very conservative approach
  - assures no deadlock
  - may have high efficiency penalty
- detection
  - deadlock can occur
  - runs expensive detection algorithm
  - deals with any detected deadlock
- avoidance – strategy in the middle
  - focuses on condition (4) and makes wise choices
Deadlock Avoidance

- requires knowledge of future resource requirements
- 2 ways to do avoidance:
  1. process initiation denial: don’t start a process if its demands might lead to deadlock
  2. resource allocation denial: don’t grant incremental resource request if that allocation might lead to deadlock

let $R$ be the resource vector: $[R_1 \ R_2 \ R_3 \ \ldots \ R_m]$ total amount of each resource in system

let $V$ be the available vector: $[V_1 \ V_2 \ V_3 \ \ldots \ V_m]$ total amount of each resource not allocated to a process
Deadlock Avoidance

Let C be the claim matrix: maximum requirements of each process for each resource:

\[
\begin{bmatrix}
C_{11} & \cdots & C_{1m} \\
C_{21} & \cdots & C_{2m} \\
C_{31} & \cdots & C_{3m} \\
\vdots & \ddots & \vdots \\
C_{n1} & \cdots & C_{nm}
\end{bmatrix}
\]

\(C_{ij}\) is requirement of process \(i\) for resource \(j\)

Deadlock Avoidance

Let A be the allocation matrix:

\[
\begin{bmatrix}
A_{11} & \cdots & A_{1m} \\
A_{21} & \cdots & A_{2m} \\
A_{31} & \cdots & A_{3m} \\
\vdots & \ddots & \vdots \\
A_{n1} & \cdots & A_{nm}
\end{bmatrix}
\]

Currently allocated resources

\(A_{ij}\) is quantity of resource \(j\) now allocated to process \(i\)
Deadlock Avoidance

- with these vectors and matrices, we have:
  - all resources are either allocated or available:
    \[ R_i = V_i + \sum_{k=1}^{n} A_{ki} \quad \text{for all } i \]
  - no process can claim more than the total amount of resources now available:
    \[ C_{ki} \leq R_i \quad \text{for all } k,i \]
  - no process is allocated more resources of any type than it originally claimed it needed:
    \[ A_{ki} \leq C_{ki} \quad \text{for all } k,i \]

Deadlock Avoidance

- OK to start a new process iff:
  - i.e., begin only if maximum claim of all current n processes plus those of the new process can be met
    \[ R \geq \sum_{k=1}^{n} C_{k*} + C_{(n+1)*} \]
  - ⇒ process initiation denial
    - always assumes worst case (everyone wants everything at once), so may needlessly delay a process
Banker’s Algorithm

- Djikstra, 1965
- a resource denial algorithm
- consider a system with:
  - fixed number of processes
  - fixed number of resources

- a state of this system is the current allocation of resources as represented by R, V, C and A
- a state is:
  - safe: there is $\geq 1$ order in which all procs can be run to completion without deadlock
  - unsafe: there is no order in which all procs ...

- idea: when a proc asks for a resource, see if granting that resource would result in a safe or unsafe state
  - block requesting proc if result would be unsafe
Banker’s Algorithm

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>P1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>P2</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Claim matrix | Allocation matrix

is this a safe state?  (example from Stallings)

can P1 finish?
  • no: cannot grant all its resource requests
### Banker’s Algorithm

<table>
<thead>
<tr>
<th>Claim matrix</th>
<th>Allocation matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

**can P2 finish?**
- **yes:** grant 1 more R3 then P2 runs to completion releasing $6 \times R1$, $1 \times R2$ and $3 \times R3$

---

### Banker’s Algorithm

<table>
<thead>
<tr>
<th>Claim matrix</th>
<th>Allocation matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

**with P2 finished,**
- **P1** now can be granted its requests and finish, releasing $3 \times R1$, $2 \times R2$ and $2 \times R3
Banker’s Algorithm

<table>
<thead>
<tr>
<th>Claim matrix</th>
<th>Allocation matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
</tr>
</tbody>
</table>

with P2 and P1 finished,
• P3 now can be granted its requests and finish, releasing $3 \times R1$, $1 \times R2$ and $4 \times R3$

with P2, P1, and P3 finished,
• P4 now can be granted its requests and finish, releasing $4 \times R1$, $2 \times R2$ and $2 \times R3$

every process now finished, so, this is a safe state
### Banker’s Algorithm

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Claim matrix

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Allocation matrix

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>9</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

#### Claim matrix

Would granting P1’s request for $1 \times R1$ and $1 \times R3$ result in a safe state?

#### Allocation matrix

If we do the grant we get this: is this a safe state?
Banker’s Algorithm

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Claim matrix

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Allocation matrix

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>110</td>
<td>112</td>
<td>115</td>
</tr>
<tr>
<td>639</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

V

R

if we do the grant we get this: is this a safe state?

no: no process can complete

e.g., all need more R1’s but there are none
and none can be freed

Avoidance with Banker’s Algorithm

- deadlock avoidance = staying in a safe state

- Banker’s algorithm does not predict deadlock with certainty
  - an unsafe state is not guaranteed to produce deadlock
  - an unsafe state has the potential to result in deadlock: that is what Banker’s avoids
Avoidance with Banker’s Algorithm

- costs of using Banker’s:
  1. maximum resource requirements for each process must be known and stated in advance
  2. processes must be independent: order in which they run is unimportant
Avoidance with Banker’s Algorithm

- costs of using Banker’s:
  1. maximum resource requirements for each process must be known and stated in advance
  2. processes must be independent: order in which they run is unimportant
  3. must be a fixed number of both resources and processes

Observation on Deadlock Handling

- Tanenbaum notes:
  “…the schemes described earlier under the name ‘prevention’ are overly restrictive, and the algorithm described here as ‘avoidance’ requires information that is usually not available. If you can think of a general–purpose algorithm that does the job in practice as well as in theory, write it up and send it to your local computer science journal.”

- there is one other approach…
Ostrich Algorithm

- stick your head in the sand and pretend there’s no such thing as deadlock
- practical:
  - no cost for memory or processing overhead
  - deadlock is a sufficiently rare occurrence
- UNIX uses this approach
  - no detection, avoidance or prevention
  - so deadlocks can happen
- suitable for settings where deadlocks are rare and can be tolerated even when they occur

Programs

- how do you run a program?
Programs

- how do you run a program?
- program exists as disk file
  - called a load module
  - contains executable code
- how get file contents into memory?
  - need an OS service that loads load modules into memory: the loader
  - loader loads load modules into memory
Where do load modules come from?

- evolution of a load module (e.g., for C programs):
  - source code
  - header files
  - library object files
  - compiler
  - assembler code
  - assembler
  - linker
  - object code
  - load module

Building a Load Module: CPP

```
/usr/lib/gcc-lib/i586-mandrake-linux-gnu/2.96/cppl
-lang-c
-v
-D__GNUC__=2
-D__GNUC_MINOR__=96
-D__GNUC_PATCHLEVEL__=0
-D__ELF__
-D_unix
-D_linux
-D_i386
-D__i386__
-D__tune_pentium__
-DSystem(posix)
-D__NO_INLINE__
-Asystem(i386)x
-Acpu(i386)x

helloworld.c /tmp/ccmEBvF0.i
```

GNU CPP version 2.96 20000731
(Mandrake Linux 8.2 2.96-0.76mdk)
(cpplib) (i386 Linux/ELF)
ignoring nonexistent directory
"/usr/i586-mandrake-linux-gnu/include"
#include "...
#include <...>

End of search list.
Building a Load Module: CC1

/usr/lib/gcc-lib/i586-mandrake-linux-gnu/2.96/cc1
/tmp/ccmEBvF0.i
-quiet
-dumpbase
helloworld.c
-version
-o /tmp/ccE5t4jQ.s

GNU C version 2.96 20000731 (Mandrake Linux 8.2 2.96-0.76mdk)
i586-mandrake-linux-gnu) compiled by GNU C version 2.96
20000731 (Mandrake Linux 8.2 2.96-0.76mdk).

Building a Load Module: AS

as
-V
-Qy
-o /tmp/ccugRXOH.o
/tmp/ccE5t4jQ.s

GNU assembler version 2.11.92.0.12 (i586-mandrake-linux-gnu)
using BFD version 2.11.92.0.12 20011121
Building a Load Module: **link**

```bash
/usr/lib/gcc-lib/i586-mandrake-linux-gnu/2.96/collect2
   -m elf_i386
   -static
   -o helloworld
/usr/lib/gcc-lib/i586-mandrake-linux-gnu/2.96/../../../crt1.o
/usr/lib/gcc-lib/i586-mandrake-linux-gnu/2.96/../../../crti.o
/usr/lib/gcc-lib/i586-mandrake-linux-gnu/2.96/crtbegin.o
-L/usr/lib/gcc-lib/i586-mandrake-linux-gnu/2.96
-L/usr/lib/gcc-lib/i586-mandrake-linux-gnu/2.96/../../../
tmp/ccugRXOH.o
-lgcc
-lc
-lgcc
/usr/lib/gcc-lib/i586-mandrake-linux-gnu/2.96/crtend.o
/usr/lib/gcc-lib/i586-mandrake-linux-gnu/2.96/../../../crtn.o
```

---

**Evolution of a load module**

- start with simple:
  ```c
  main() {
    printf("Hello world\n");
  }
  ```

- length: 44 bytes
Evolution of a load module

- compiler produces (partial listing):
  ```
  .string "Hello world\n"
  .text
  .globl main
  .type main, @function
  main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  subl $12, %esp
  pushl $.LC0
  call printf
  addl $16, %esp
  movl %ebp, %esp
  popl %ebp
  ret
  ```

- on Linux platform
- file size: 399 bytes

- assembler generates object file
  - on Linux, file size is 936 bytes
  - on Sun Sparc (Solaris), file size is 784 byte

- in the .o file, we find (as reported by nm command):
  ```
  00000000 T main
  U printf
  ```

  printf is an Undefined symbol in this object file
  main is a defined symbol in this object file Text segment
Evolution of a load module

- linker’s job: to resolve such undefined references by linking library object code with our newly created object code
  - or making such linking possible in future...
- `printf()` is in C run-time library (on Linux, e.g., `libstdc++-3-libc6.2-2-2.10.0.a`)
- linker generates load module:
  - on Linux, file size is 1,727,238 bytes
  - on Sun Sparc (Solaris), file size is 374,264 bytes

Different Kinds of Load Module

- load modules are built to suit the way the loader loads them into memory
- absolute:
  - real address used to refer to
    - locations holding variable values, e.g., CLR 05A21BE4
    - jump targets, e.g., JMP 4AF0328C
  - program must be loaded into real memory at exactly the location it was built for
Different Kinds of Load Module

- **absolute**
  - programmer may need to know or use real addresses in source code
  - requires compiler to know real addresses at compile time
  - program may run no matter where loaded, but will only work correctly when loaded at “it’s” address
  - almost impossible with multi-user/multi-tasking systems
Different Kinds of Load Module

- relocatable:
  - solution: delay that association until load time
  - generate code assuming module starts at location 0
  - compute all memory references relative to this start location
  - loader “fixes” relative-to-0 references when loading
    - requires extra information provided by compiler and/or assembler
    - produces a load module ‘bound’ to the location where it is loaded

- so if load module swapped out, then back in to different address: badness
- problem: associated a real memory address to variables, targets at load time
Different Kinds of Load Module

- dynamic run-time:
  - solution: delay that association until run-time
  - load program into real memory keeping relative addresses as they are
  - absolute addresses are calculated as needed at run time
    - using special hardware (to be talked about later)

Linkers and Loaders

- highly inter-related
- no linking needed when:
  - all source code contained in module (i.e., no external references)
  - very rarely possible or desireable
Static Linking

- after assembly (with `.o` file):
  - relative object code for each invoked function is appended to code generated from assembler
  - linkage done on concatenated set of object components
  - produces one self-contained blob of code
    - may be big

Linkage

before:

```
00000000 _main: clr r0
00001044 jsr _printf
00002AE0 ret ...
0000000 _printf: mov r0,-(sp)
00000000 ret
```

```
Dynamic Linking

- link at load time
  - load module contains unresolved external references
  - locate and link object code for needed functions as load module is loaded into memory
  - adjust as needed while loading, relative to overall load module start
Dynamic Linking

- link at run time
  - call to function not already linked-in causes that function to be linked at that moment
  - e.g., `printf()` would be linked only when it is actually invoked

Static Linking: Pros and Cons

- fully self-contained
- immune to library changes
- more portable

- very large executable image
- change to program ⇒ full re-link of everything
Dynamic Linking (load time): Pros and Cons

- benefits from new or updated libraries without re-link
- sharing of libraries in memory
- not fully self-contained
- less portable: needs libraries to be available
- may not work with updated libraries

Dynamic Linking (run time): Pros and Cons

- benefits from new or updated libraries without re-link
- sharing of libraries in memory
- smaller executable image: only has functions actually used
- not fully self-contained
- less portable: needs libraries to be available
- may not work with updated libraries
- performance degradation on first “missed” invocation